Отчет по курсовой работе "Построение нейронной сети Кохонена"

КМБО-01-21, Рустем Сиразетдинов

9 декабря 2023 г.

Оглавление

Глава 1

Предисловие

В современном мире работа исследователей все чаще направлена на изучение и развитие *искусственного интеллекта* (далее - ИИ). Работы в этой области обладают значительным прикладным потенциалом. Но уже сейчас их применение крайне разнообразно: предсказание заболеваний на основе анализов человека, распознавание биометрических данных, обработка больших данных и предсказания на их основе, голосовые ассистенты, компьютерное зрение и тд. Особое место занимает генеративный искусственный интеллект, создающий, например, мультимедийные материалы или тексты, неотличимые от результатов человеческих усилий.

Все это результаты ученых, пытающихся создать структуры подобные человеческой нервной системе в цифровом пространстве. Мы, конечно, имеем ввиду *искусственные нейронные сети* (далее - нейронные сети) - одна из самых популярных и успешных реализаций ИИ, воссоздающая нервную систему человека лишь с некоторыми упрощениями. Конечно, здесь невозможно не упомянуть и огромную работу нейробилогов, предоставляющих информацию о нервной системе человека, а также все серьезные проблемы урегулирования развития ИИ, однако оставим их рассмотрение за рамками текущего отчета.

В этой работе мы постараемся дать некоторые начальные положения устройства и работы нейронных сетей, более подробно рассмотрим саморегулирующиеся сети, что представляют класс сетей с обучением без подкрепления, приведем реализацию такой нейронной сети и продемонстрируем ее работу на примере задачи распознавания цифр.

Глава 2

Введение

В данной работе читателю предлагается сначала познакомиться с таким типом нейронных сетей как самоорганизующаяся карта Кохонена, а после перейти к ее реализации на языке *C++* в стандарте **ISO International Standard ISO/IEC 14882: 2020(E) – Programming Language C++**. При компиляции кода программы были использованы следующие флаги.

```
clang++ -pedantic-errors -Wall -Werror -g3 -O0 --std=c++20 -g
```

Программа прошла проверку на утечки памяти с помощью утилиты *valgrind* с флагом `-s`.

Работа представлена шестью главами. В первой главе описана мотивации работы. Во второй мы знакомим читателя с общими положениями нейронных сетей. В главе три мы рассматриваем такой тип нейронных сетей как самоорганизующиеся карты Кохонена. В четвертой главе приводим реализацию модели нейронной сети Кохонена, введенной нами в главе три. Четвертая глава описывает полученные нами результаты и подытоживает всю работу. В главе пять представлен подробный код программы, написанной нами в главе четыре, а глава шесть представлена используемой при написании работы литературой.

2.1 Знакомство с нейронными сетями

Исследования по нейронным сетям связаны с тем, что способ обработки информации человеческим мозгом в корне отличается от методов, применяемых обычными цифровыми компьютерами. Мозг представляет собой сложный, нелинейный, параллельный компьютер. Важная его особенность заключается в организации своих структурных компонент, называемых *нейронами*, так, чтобы они могли выполнять конкретные задачи во много раз быстрее, чем могут позволить самые быстродействующие современные компьютеры. Примером такой задачи обработки информации служит обычное *зрение*. В функции зрительной системы входит создание представления окружающего мира в таком виде, который обеспечивает возможность взаимодействия с этим миром. Т.е. мозг последовательно выполняет ряд задач распознавания и на это ему требуется порядка 100-200 миллисекунд, в то время как выполнение аналогичных задач даже меньшей сложности на компьютере может занять несколько дней.

Что позволяет мозгу достичь такого результата? Дело в том, что при рождении мозг человека крайне *пластичен*, т.е. легко позволяет нервной системе подстраиваться (настраивать нейроны и синопсисы) под окружающую среду на основе накапливающегося опыта. Аналогично, в искусственных нейронных сетях работа проводится искусственными нейронами, которые прошли процесс *обучения*. Процедура, использующаяся для процесса обучения, называется *алгоритмом обучения*. Ее задача состоит в выстраивании в определенном порядке синаптических весов сети для обеспечения правильных взаимосвязей между самими нейронами. Таким образом мы пришли к следующему определению искусственной нейронной сети.

> **Нейронная сеть** - это распределенный параллельный процессор, состоящий из элементарных единиц обработки информации, накапливающих экспериментальные знания и предоставляющих их для последующей обработки.

2.2 Модели нейронов

Нейрон представляет собой единицу обработки информации в нейронной сети. На рис. 2.1 показана модель нейрона, лежащего в основе нейронных сетей. В модели можно выделить три основных элемента.

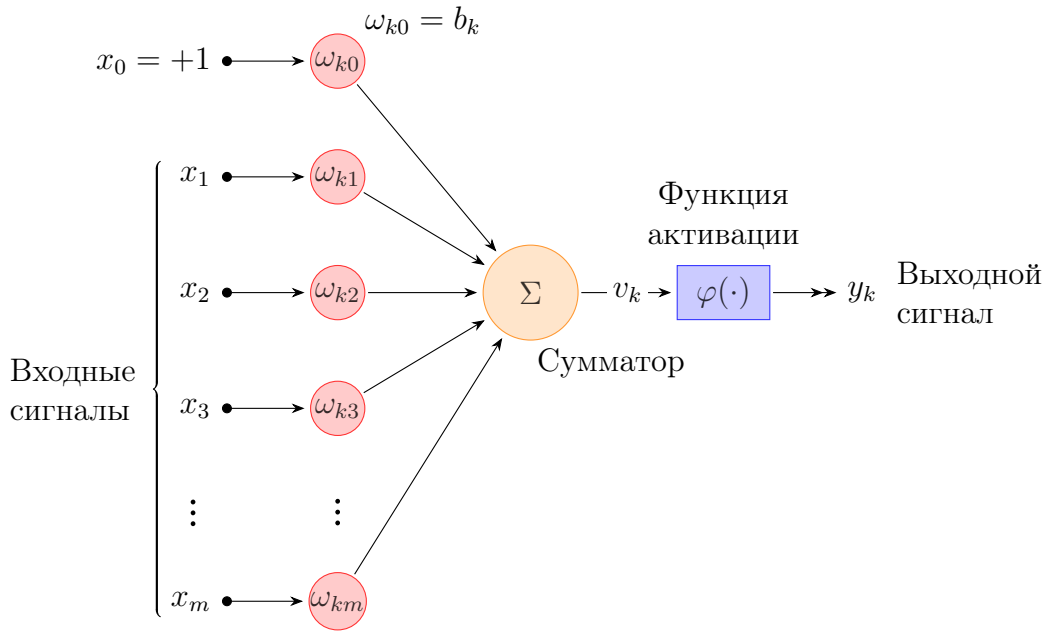


Рис. 2.1: Нелинейная модель нейрона

1. Набор *синапсов*, каждый из которых характеризуется своим весом. В частности сигнал x_j на входе синапса j, связанного с нейроном k, умножается на вес ω_{kj}. Синаптический вес нейрона может иметь как положительные, так и отрицательные значения.

2. *Сумматор* складывает входные сигналы, взвешенные относительно соответствующих синапсов нейрона. Эта операция представляет собой линейную комбинацию.

3. *Функция активации* ограничивает значения выходного сигнала нейрона, так чтобы он лежал в диапазоне $[0, 1]$ или $[-1, 1]$.

Очень часто в модель нейрона включают и пороговый элемент, обозначенный на рис. 2.1 символом b_k. Эта величина отражает увеличение или уменьшение входного сигнала v_k, подаваемого на функцию активации. Другими словами, использование порога b_k обеспечивает эффект аффинного преобразования выхода линейного сумматора u_k см. рис. 2.2.

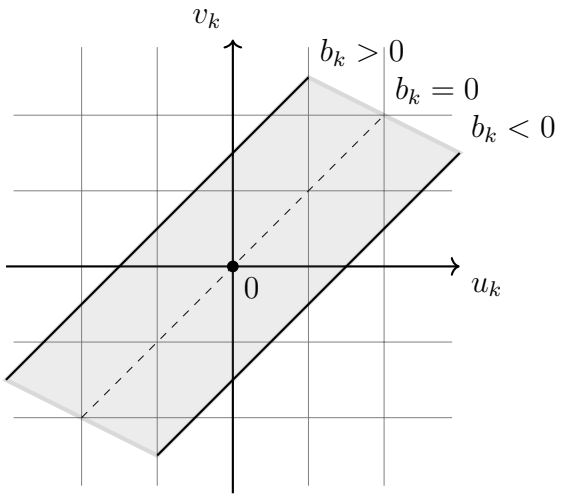


Рис. 2.2: Аффинное преобразование, вызванное наличием порогового элемента b_k.

В математическом представлении функционирование нейрона k можно описать следующей парой уравнений:

$$\begin{aligned} u_k &= \sum_{j=1}^{m} \omega_{kj} x_j, \\ y_k &= \varphi(v_k),\ v_k = u_k + b_k \end{aligned} \tag{2.1}$$

где $x_1, x_2, \ldots, x_m$ - входные сигналы; $\omega_{k1}, \omega_{k2}, \ldots, \omega_{km}$ - синаптические веса нейрона k; u_k - линейная комбинация входных сигналов; b_k - величина порогового элемента; $\varphi(\cdot)$ - функция активации; y_k - выходной сигнал нейрона.

2.3 Типы функций активации

Функции активации определяют выходной сигнал нейрона в зависимости от входного сигнала v. Прежде чем идти дальше, давай обсудим мотивация введения функции активации.

Как отмечалось ранее, на вход некоторому нейрону k подается сигнал u_k, являющийся линейной комбинацией выходных сигналов "прошлых"нейронов, а именно входного слоя (подробнее о слоях нейронов будет рассказано в разделе Представление нейронных сетей с помощью направленных графов). При отсутствии функции активации, выходное значения y_k по прежнему будет линейной комбинацией нейронов входного слоя, а значит и выход всей нейронной сети будет линейной комбинацией. Но тогда для каждого набора входных сигналов мы будем получать всю ту же

линейную комбинацию, что привод к потере смысла использования всякой нейронной сети, ведь она как и человеческая нервная система должна работать по разному для каждого набора входных сигналов. Чуть позже, когда будем обсуждать тему способов обучения нейронной сети снова убедимся в важности функции активации.

Теперь перейдем к классификации. Можно выделить три основных типа функции активации.

1. *Функция единичного скачка*, или пороговая функция. Этот тип показан на рис. 2.3a и описывается следующим образом:

$$\varphi(v) = \begin{cases} 1, & \text{если } v \geq 0; \\ 0, & \text{если } v < 0; \end{cases} \tag{2.2}$$

   Соответственно выходной сигнал нейрона k такой функции можно представить как

$$y_k = \begin{cases} 1, & \text{если } v_k \geq 0; \\ 0, & \text{если } v_k < 0; \end{cases} \qquad v_k = \sum_{j=1}^{m} \omega_{kj} x_j + b_k. \tag{2.3}$$

2. *Кусочно-линейная функция.* Кусочно-линейная функция показанная на рис. 2.3b, описывается следующим выражением:

$$\varphi(v) = \begin{cases} 0, & v \leq -0.5, \\ \beta|v|, & 0.5 > v > -0.5; \\ 1, & v \geq 0.5; \end{cases} \tag{2.4}$$

   где коэффициент β усиления в линейной области предполагается равным единице.

3. *Сигмоидальная функция.* Сигмоидальная функция, график, которой изображен на рис. 2.3c, является наиболее распространенной для создания нейронных сетей. Она поддерживает баланс между линейным и нелинейным поведением. Примером сигмоидальной функции может служить логистическая функция (рис. 2.3d), задаваемая выражением:

$$\varphi(v) = \frac{1}{1 + \exp(-\alpha v)}, \tag{2.5}$$

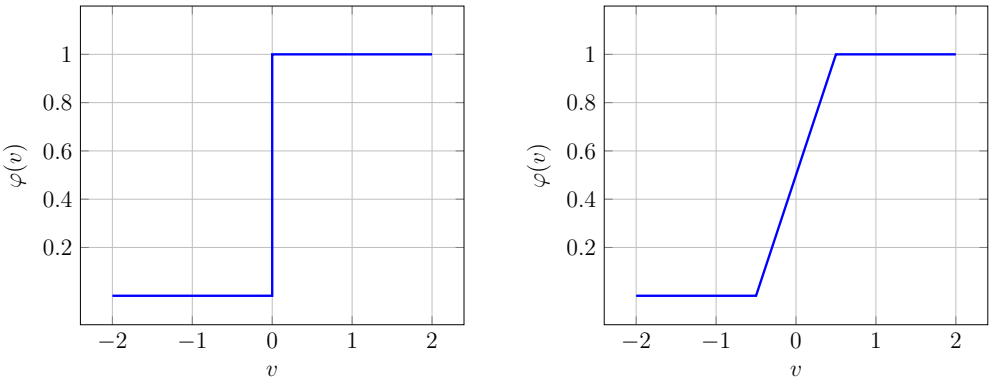
   где α - параметр наклона сигмоидальной функции, отвечающий за то, как быстро или медленно функция будет возрастать в линейной своей части.

Мы рассмотрели несколько примеров функций активации, у каждой из них область значений представляется отрезком $[0, 1]$. Однако иногда требуется функция активации, имеющая область значений отрезок $[-1, 1]$. В этом случае мы хотим от нее симметричности относительно начала координат, в частности, мы может определить пороговую функцию (рис. 2.4a):

$$\varphi(v) = \operatorname{sgn}(v) = \begin{cases} -1, & v < 0, \\ 0, & v = 0, \\ 1, & v > 0. \end{cases}$$

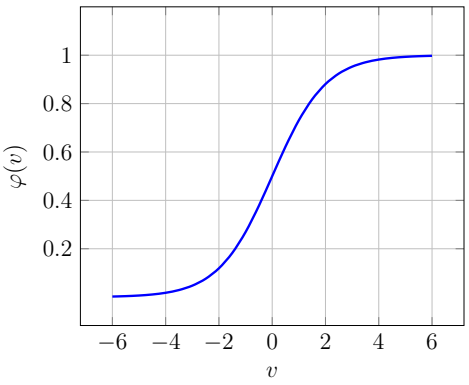
Еще отличным примером сигмоидальной функции будет служить гиперболический тангенс (рис. 2.4b):

$$\varphi(v) = \tanh(v)$$

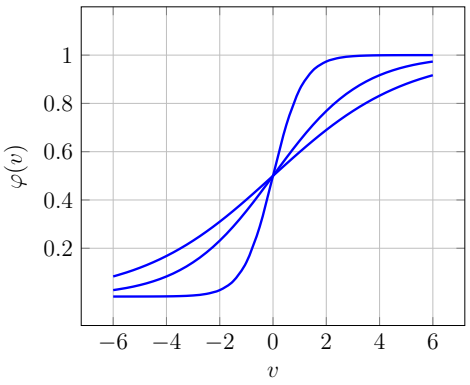


(a) Пороговая функция

(b) Кусочно-линейная функция

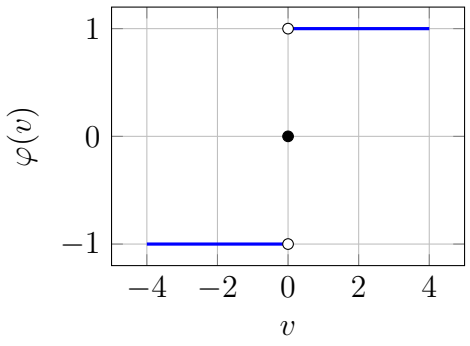


(c) Сигмоидальная функция

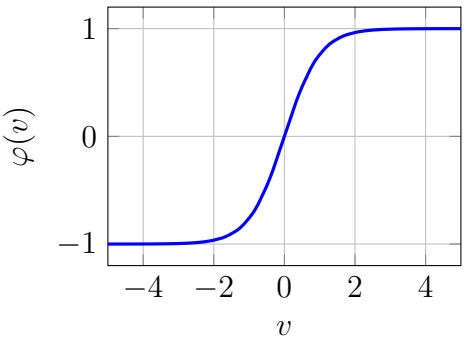


(d) Логистическая функция

Рис. 2.3: Разных типов функции активации



(a) Сигнум



(b) Гиперболический тангенс

Рис. 2.4: Разных типов функции активации (продолжение)

2.4 Представление нейронных сетей с помощью направленных графов

Блочная диаграмма, представленная на рис. 2.1, обеспечивают функциональное описание различных элементов, из которых состоит модель искусственного нейронов. Эту модель можно упростить, если использовать идею, которая согласно [4] называется граф передачи сигналов.

Граф передачи сигнала - это сеть направленных связей, соединяющих

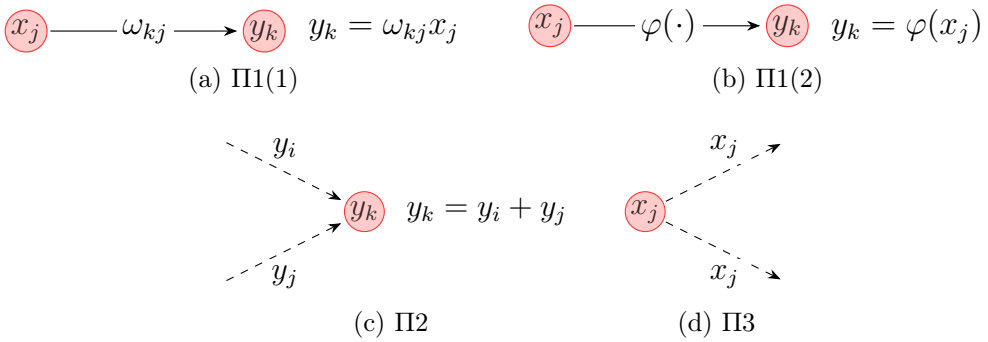
отдельные точки(узлы). С каждым узлом j связан сигнал x_j. Обычная направленная связь начинается в некотором узле j и заканчивается в другом узле k. С ней связана некоторая *передаточная функция*, определяющая зависимость сигнала y_k узла k от сигнала x_j узла j. Причем прохождение сигнала по частям графа подчиняется трем основным правилам.

П1 Направление прохождения сигнала вдоль каждой связи определяется направлением стрелки. При этом можно выделить два типа связей.

- *Синаптические связи.* Их поведение определяется линейным соотношением вход-выход. А именно, как показано на рис. 2.5a , сигнал узла x_j умножается на синаптический вес ω_{kj}, в результате чего получается сигнал узла y_k.
- *Активационные связи.* Их поведение определяется нелинейным соотношением вход-выход. Этот вид связи показан на рис. 2.5b , где $\varphi(\cdot)$ - нелинейная функция активации.

П2 Сигнал узла равен сумме сигналов, поступающих на его вход, см. рис. 2.5c.

П3 Сигнал данного узла передается по каждой исходящей связи без учета передаточных функций исходящих связей, см. рис. 2.5d.



(a) П1(1) (b) П1(2) (c) П2 (d) П3

Рис. 2.5: Основные правила построения графов передачи сигналов

На рис. 2.6 показан пример графа передачи сигнала. Эта модель нейрона, соответствующая блочной диаграмме, приведенной на рис. 2.1, но проще с виду, хотя и содержит все функциональные детали.

Принимая в качестве модели нейрона граф передачи сигнала, показанный на рис. 2.6, можно сформулировать еще одно определение нейронной сети.

Нейронная сеть - это направленный граф, состоящий из узлов, соединенных синаптическими и активирующими связями, который характеризуется следующими четырьмя свойствами.

1. Каждый нейрон представляется множеством линейных синаптических связей, внешним порогом и, возможно, нелинейной связью активации. Порог, представляемый входной синаптической связью, считается равным $+1$.

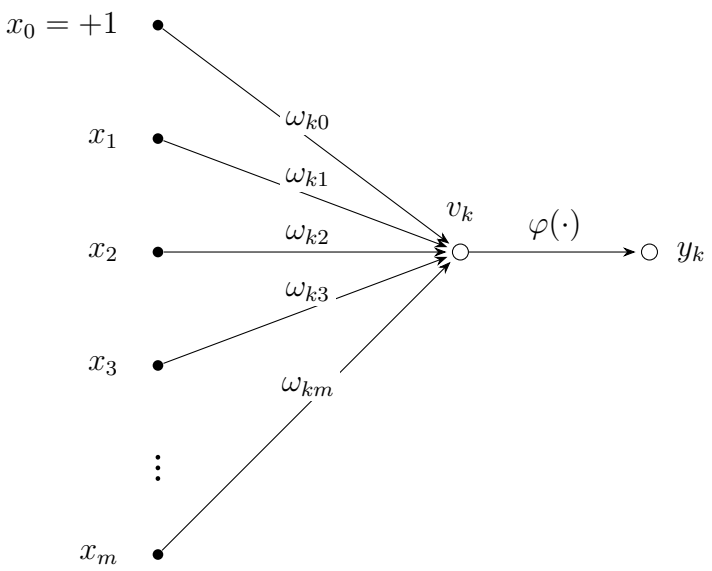


Рис. 2.6: Граф передачи сигнала для одного нейрона

2. Синаптические связи нейрона используются для взвешивания соответствующих входных сигналов.

3. Взвешенная сумма входных сигналов определяет индуцированное локальное поле (v_k) каждого конкретного нейрона.

4. Активационные связи модифицируют индуцированное локальное поле нейрона, создавая выходной сигнал.

Направленный граф, определенный указанным выше способом, является *полным*, т.е. он описывает не только прохождение сигнала между нейронами, но и передачу сигнала в самом нейроне. Часто нам будет нужно описывать только прохождение сигнала между нейронами, не упоминая о второй части. В этих случаях можно использовать сокращенную форму этого графа - *частично полную*, в таком случае граф будет называться *архитектурным*. Архитектурный граф (см. рис. 2.7) описывает структуру нейронной сети и по определению обладает следующими свойствами.

1. Входные сигналы графа формируются узлами источника или входными элементами.

2. Каждый нейрон представляется одним узлом, который называется вычислительным.

3. Линии передачи сигнала, соединяющие узлы-источники и вычислительные узлы графа, не имеют веса, а только лишь определяют направление прохождения сигнала на графе.

В случае многослойных сетей прямого распределения узлы источника входного слоя сети формируют входной вектор, поступающий на нейроны второго слоя (т.е. первого скрытого слоя). Выходные сигналы второго слоя используются в качестве входных для третьего и т.д. Обычной нейроны каждого из слоев сети используют в качестве входных сигналов выходные сигналы нейронов только предыдущего слоя. Набор выходных сигналов нейронов выходного (последнего) слоя сети определяет

общий отклик сети на данный входной вектор (образ), сформированный узлами источника входного (первого) слоя.

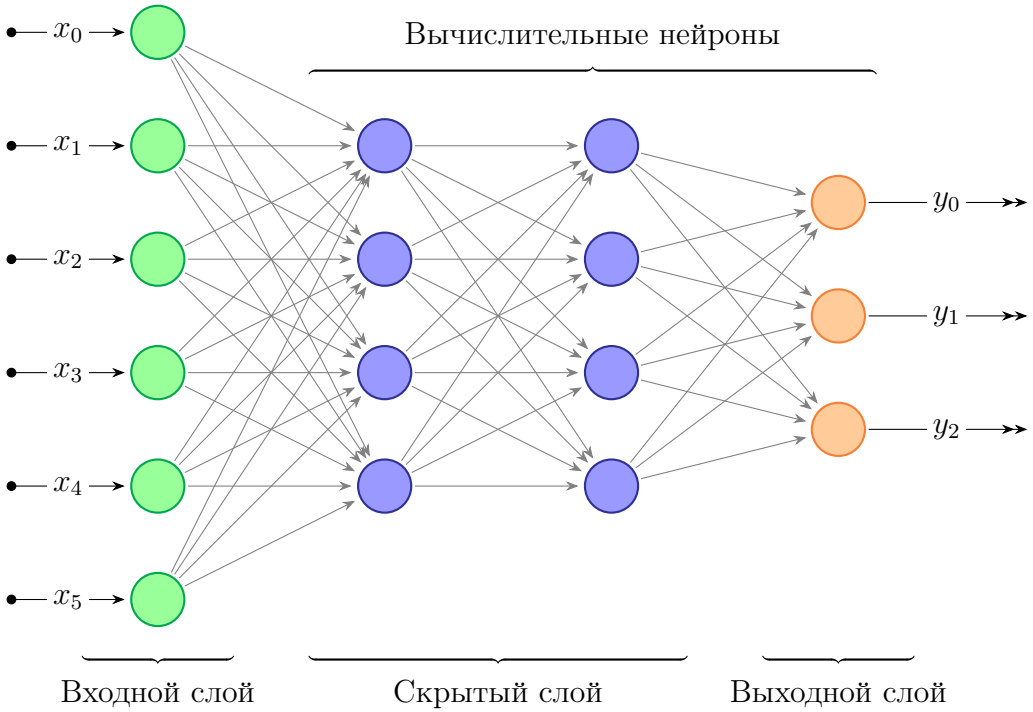


Рис. 2.7: Архитектурный графа

Мы вполне можем считать, что на рис. 2.7 изображена нейронная сеть $6-4-4-3$, т.к. она имеет 6 входных, 2 слоя по 4 скрытых и 2 выходных нейрона. Если все узлы каждого конкретного слоя соединены со всеми узлами смежных слоев, то такую нейронную сеть принято называть *полносвязной* иначе *неполносвязной*.

2.5 Процесс обучения

Самым важным свойством нейронных сетей является их способность обучаться на основе данных окружающей среды и в результате обучения повышать свою производительность в соответствии с определенными правилами. Обучение нейронной сети происходит через интерактивную корректировку синаптических весов и порогов. В идеальном случае нейронная сеть получает знания об окружающей среде на каждой итерации процесса обучения. С точки зрения нейронных сетей можно дать следующее определение процесса обучения.

> *Обучение* - это процесс, в котором свободные параметры нейронной сети настраиваются посредством моделирования среды в которую эта сеть встроена. Тип обучения определяется способом настройки этих параметров.

Такое определение процесса обучения подразумевает последовательное выполнение следующих событий.

1. В нейронную сеть поступают *стимулы* из внешней среды.

2. В результате этого изменяются свободные параметры нейронной сети.

3. После изменения внутренней структуры нейронная сеть отвечает на возбуждения уже иным образом.

Мы отметим, что создание и использование алгоритма обучения есть дело творческое, ибо не существует универсального алгоритма обучения, подходящего для всех архитектур нейронных сетей. Существует лишь набор средств, представленный множеством алгоритмов обучения, каждый из которых имеет свои достоинства.

Сейчас мы не будем рассматривать разные алгоритмы обучения поскольку это весьма объемный и трудоемкий процесс, однако далее при рассмотрении особой архитектуры нейронной сети "самоорганизующаяся карта Кохонена являющейся главной темой данной работы, мы обсудим алгоритм обучения (конкурентный подход без учителя) применяемый для таких нейронный сетей, а пытливый читатель может прочитать о других алгоритмах и *парадигмах* обучения в [4] глава 2.

2.6 Алгоритмы и нейронные сети

До появления науки о нейронных сетях уже была существенно развита теория алгоритмов, которая успешно использовалась учеными и разработчиками того времени, так какие недостатки алгоритмов пришлось закрывать созданием нейронных сетей?

Давайте рассмотрим, какие задачи выполняет алгоритм? Сам по себе каждый алгоритм, конечно, является набором инструкций для вычислительной машины, помимо этого, он призван решить конкретную проблему, например, набор инструкций для сортировки значений контейнера, увеличение скорости обращения к объектам с помощью кэширования, создания отказоустойчивости и надежности с помощью алгоритмов создания журналов, документации о релевантных событиях. Однако, кроме этого, мы ожидаем от алгоритмов эффективности в работе, например, качество и скорость сжатия данных, т.е. не каждый набор инструкций мы определим как алгоритм. Таким образом, алгоритм обеспечивает систематический и логический подход к решению задачи.

Во внутреннем устройстве нейронных сетей тоже участвуют алгоритмы, но тогда чем же алгоритмы и нейронные сети отличаются? Главным и самым важным отличием и преимуществом нейронных сетей над алгоритмами является их способность к обучению - способность давать правильные и *новые* ответы на наборах данных, которые никогда ранее не встречались и улучшать свои же ответы на неоднократно протестированных входных данных. Нейронные сети в процессе обучения и работы способны обнаруживать сложные связи между изучаемыми объектами, в то время как разработчики могли и не догадываться об их существовании. При работе с алгоритмом все его множество ответов было заранее заложено в него во время проектирования.

Преимущества нейронных сетей

1. *Нелинейность*. Нелинейность является чрезвычайно важным свойством, когда сам физический механизм, отвечающий за формирование входного сигнала, тоже является нелинейным (например, человеческая речь).

2. *Отображение входной информации в выходную.* Обучение с учителем подразумевает изменение синаптических весов на основе маркированных учебных примеров. Каждый пример состоит из входного сигнала и соответствующего ему желаемого отклика. Нейронная сеть модифицирует веса для минимизации расхождений желаемого выходного сигнала и формируемого сетью. Таким образом, нейронная сеть обучается на примерах, составляя таблицу соответствий вход-выход для конкретной задачи.

3. *Адаптивность.* Нейронные сети обладают способность адаптации синаптических весов к изменениям окружающей среды. В частности, нейроны, обученные действовать в определенной среде, могут быть легко переучены для работы в условиях незначительных колебаний параметров среды. А для работы в нестационарной среде могут быть созданы нейронные сети, изменяющие синоптические веса в реальном времени.

4. *Очевидность ответа.* В контексте задачи классификации образов можно разработать нейронную сеть, собирающую информацию не только для определения конкретного класса, но и для увеличения достоверности принимаемого решения. Впоследствии эта информация может быть использована для исключения сомнительных решений.

Помимо упомянутых выше преимуществ нейронной сети стоит упомянуть и другие: отказоустойчивость, масштабируемость, контекстная информация, единообразие анализа и проектирования. Более подробно с ними читатель может ознакомиться, например, в [4].

2.7 Приложения нейронных сетей

Мы хотели бы завершить главу Введение, рассмотрев пару примеров использования нейронных сетей в реальном мире.

Распознавание аудио Показательным примером приложения нейронных сетей может быть решение задачи распознавания речи. Эта задача довольно популярна в наше время. Любой голосовой ассистент, например, Yandex Алиса, Google Assistant, Apple Siri, Samsung Assistant нуждается в модуле распознавания речи, чтобы воспринимать голосовые команды пользователя. Кроме этого модуль распознавания аудио незаменим для пресечения незаконных попыток использования чужой музыки. Наиболее часто для решения задачи распознавания аудио используются рекурсивные или сверточные сети. В виде входной информации нейронная сеть принимает аудио спектрограмму, пример такой спектрограммы изображен на рис. 2.8

Компьютерное зрение Другим и крайне важным приложением нейронных сетей является компьютерное зрение. Две наиболее важные использующиеся технологии это машинное обучение, а конкретно, глубокое обучение и сверточная нейронная сеть, которая помогает модели машинного обучения "вглядываться разбивая изображения на пиксели и присваивая им метки. Подобно тому как это делает человек, нейронная сеть сначала различает четкие края и простые формы, а затем заполняет информацию по мере выполнения итераций своих прогнозов. Сверточная нейронная

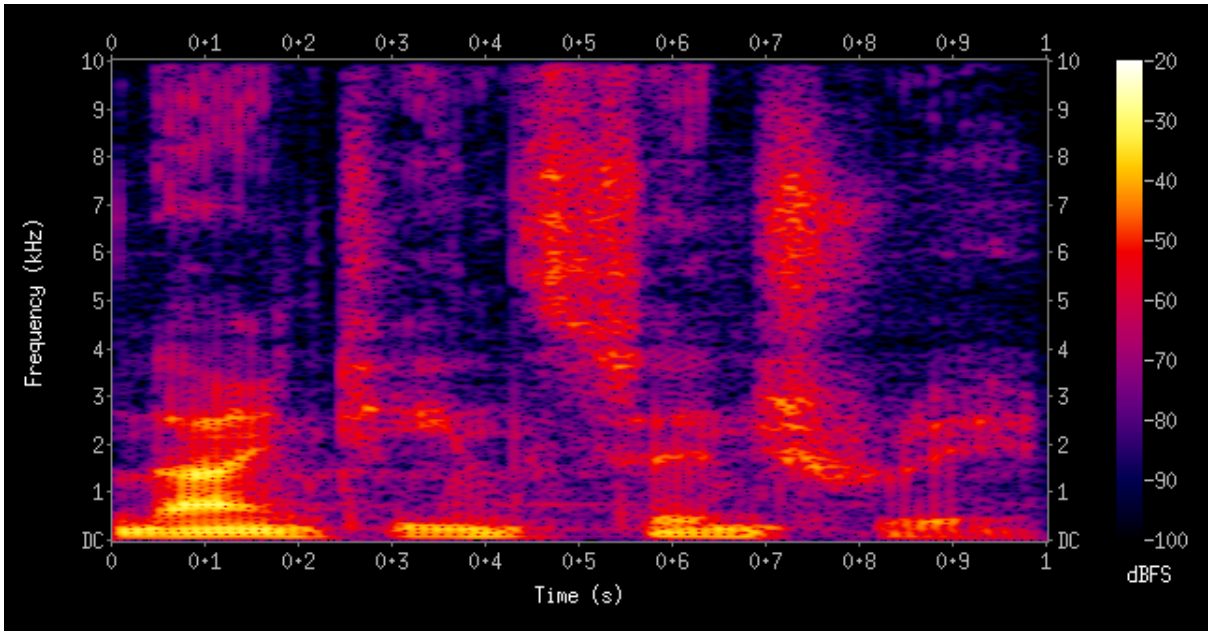


Рис. 2.8: Аудио спектрограмма спокойного произношения слов "nineteenth century" (перевод на рус. "девятнадцатый век").

сеть используется для обработки отдельных изображений, в то время как рекуррентная нейронная сеть аналогичным образом используется в обработке нескольких кадров связанные друг с другом. На рис. 2.9 приведен снимок того, как нейронная сеть распознает дорогу. Последовательно применяя к текущему кадру разные фильтры, она пытается выделить предмет, разбить его на пиксели и обработать эти данные на основе пройденного обучения.

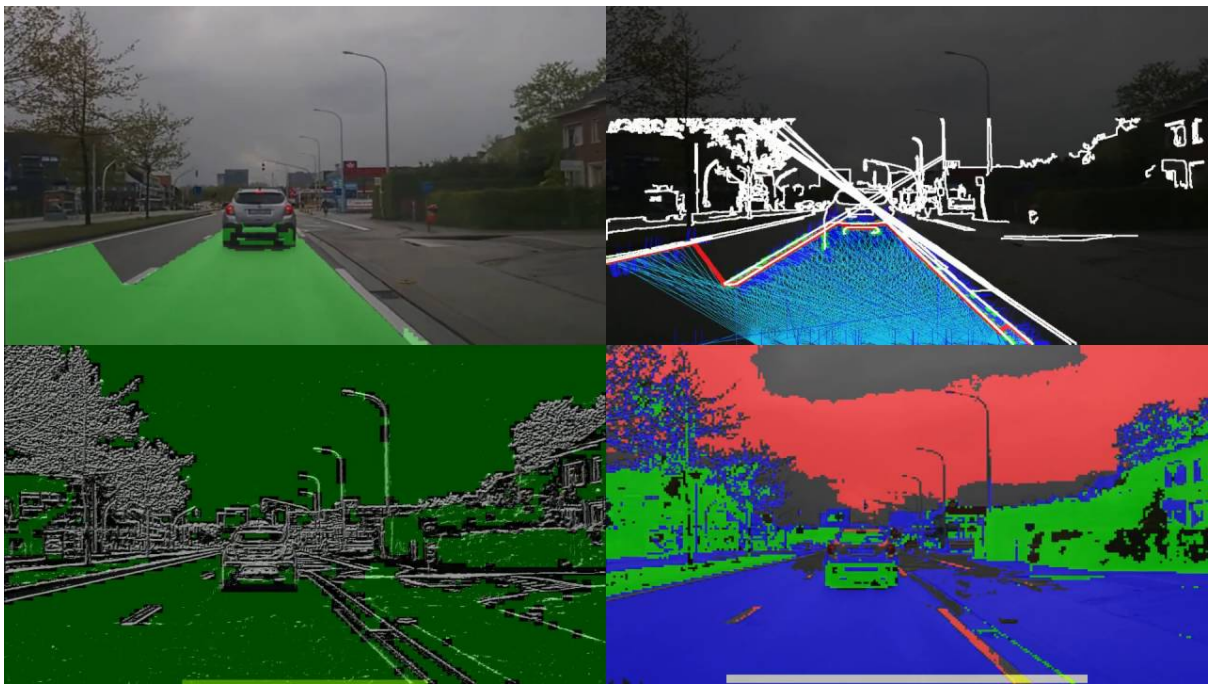


Рис. 2.9: Распознавание дороги с помощью компьютерного зрения

Глава 3

Самоорганизующиеся карты Кохонена

3.1 Введение

Карты самоорганизации - это особый класс нейронных сетей, основанный на конкурентном обучении. Нейроны выходного слоя соревнуются за право активации и в результате каждому входному сигналу (входному вектору) соответствует лишь один активированный нейрон (*нейрон-победитель*).

В картах самоорганизации нейроны помещаются в узлах решетки, обычно одно- или двухмерной. Достаточно редко можно встретить карты более высоких размерностей. Во время конкурентного процесса нейроны избирательно настраиваются на различные входные образы или классы входных образов, организуя тем самым положения нейронов-победителей, а координаты нейронов решетки являются индикаторами встроенных статистических признаков, содержащихся в примерах. Т.е. во время обучения решетка нейронов деформируется в соответствии с изменение координат нейронов, которые стремятся в точки сгущения признаков и в идеале каждому признаку соответствует некоторый нейрон в решетке, который активируется, когда во входном векторе преобладает этот признак. Отсюда берет свое происхождение и само название "самоорганизующиеся карты".

Основной целью карт самоорганизации является преобразование поступающих сигналов, имеющих произвольную размерность, в одно- или двухмерную дискретную карту (см. рис. 3.1). Помимо этого, они выполняют задачу первоначальной разведки данных, выделении признаков, сжатии, классификации или кластеризации данных перед тем как другие виды нейронных сетей обученные с учителем на выделенных с помощью самоорганизующихся карт классов признаков, начнут свою работу.

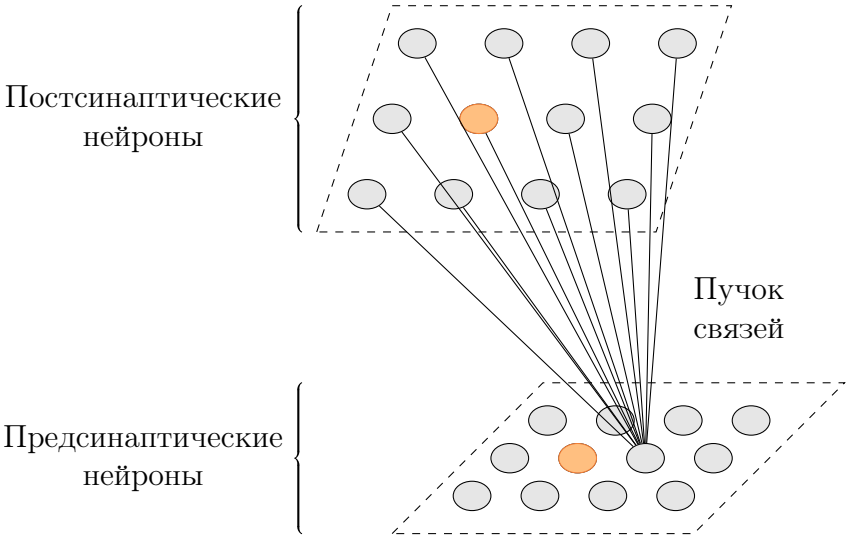
3.2 Структура сети и ее обучение

В разделе 3.1 была введена идея работы самоорганизующейся карты и теперь можно сделать следующий важный вывод.

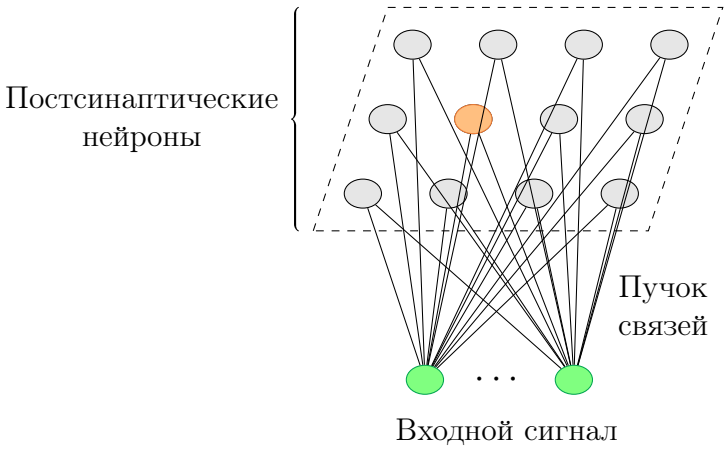
> Пространственное положение выходных нейронов в топографический карте соответствует конкретной области признаков данных, выделенных из входного пространства.

Откуда следует, что нейроны, работающие с близко расположенными областями информации, также расположены достаточно близко друг к другу, таким образом взаимодействуя друг с другом посредством коротких синаптических связей.

На рис. 3.1 показаны две основные модели отображения признаков, оранжевым цветом выделены нейроны-победители. В нашей работе будет рассмотрена именно вторая - модель Кохонена, которая по сравнению с моделью Уилшоу-ван дер Мальсбурга является более общей в том смысле, что она способна осуществлять сжатие данных (т.е. уменьшение размерности входного сигнала).



(a) Модель Уилшоу-ван дер Мальсбурга



(b) Модель Кохонена

Рис. 3.1: Основные модели самоорганизующихся карт

Алгоритм, ответственный за формирование самоорганизующейся карты, начинается с инициализации синаптических весов сети. Обычно это происходит с помощью назначения синаптическим весам малых случайных значений. При таком формировании карта признаков изначально не имеет какого-либо порядка признаков. После корректной инициализации сети для продолжения формирования карты самоорганизации запускаются три следующих основных процесса.

1. *Конкуренция.* Для каждого входного образа нейроны сети вычисляют относительные значения дискриминантной функции. Эта функция является основной

для конкуренции среди нейронов.

2. *Кооперация.* Победивший нейрон определяет пространственное положения топологической окрестности нейронов, обеспечивая тем самым базис для кооперации между этими нейронами.

3. *Синаптическая адаптация.* Последний механизм позволяет возбужденным нейронам увеличивать собственные значения дискриминантных функций по отношению к входным образам посредством соответствующих корректировок синаптических весов. Корректировки производятся таким образом, чтобы отклик нейрона-победителя на последующее применение аналогичных примеров усиливался.

Процесс конкуренции Пусть m - размерность входного пространства, а входной вектор выбирается из этого входного пространства случайно и обозначается так:

$$\boldsymbol{x} = [x_1, x_2, \ldots, x_m]^T. \tag{3.1}$$

Вектор синаптических весов каждого из нейронов сети имеет ту же размерность, что и входное пространство. Обозначим синаптический вес нейрона j следующим образом:

$$\boldsymbol{w}_j = [w_{j1}, w_{j2}, \ldots, w_{jm}]^T, \ j = \overline{1, l}, \tag{3.2}$$

где l - общее количество нейронов сети. Для того, чтобы подобрать наилучший вектор весов $\boldsymbol{w}_j$, соответствующий входному вектору $\boldsymbol{x}$, нужно сравнить скалярные произведения $\boldsymbol{w}_j^T \boldsymbol{x}$ для $j = \overline{1, l}$ и выбрать наибольшее значение. При этом предполагается, что ко всем нейроном применяется некоторое значение *насыщения*, равное порогу, взятому с противоположным знаком. Это делается для того, чтобы дополнительно выделить нейрон, который должен быть активирован и избежать неограниченного наращивания синаптических весов, которое может привести к неустойчивости системы. Таким образом, выбирая нейрон с наибольшим скалярным произведением $\boldsymbol{w}_j^T \boldsymbol{x}$, мы в результате определяем местоположение, которое должно стать центром топологической окрестности возбужденного нейрона.

Применим некоторую хитрость. Максимизация скалярного произведения $\boldsymbol{w}_j^T \boldsymbol{x}$, математически эквивалентна минимизации евклидова расстояния между векторами $\boldsymbol{x}$ и $\boldsymbol{w}_j$. Тогда если использовать индекс $i(\boldsymbol{x})$ для идентификации того нейрона, который лучше всего соответствует входному сигналу $\boldsymbol{x}$, то эту величину можно определить с помощью следующего соотношения:

$$i(\boldsymbol{x}) = \underset{1 \leqslant j \leqslant l}{\operatorname{argmin}} \|\boldsymbol{x} - \boldsymbol{w}_j\| = \sqrt{\sum_{k=1}^{m} |x_k - w_{kj}|^2}. \tag{3.3}$$

Конкретный нейрон i, удовлетворяющий данному условию, называется *победившим.*

Процесс кооперации Нейрон-победитель находится в центре топологической окрестности сотрудничающих нейронов, он является возбужденным нейроном, а такие нейроны всегда стремятся возбудить пространственно близкие к нему нейроны. Дадим определение топологической окрестности победившего нейрона i, которая плавно

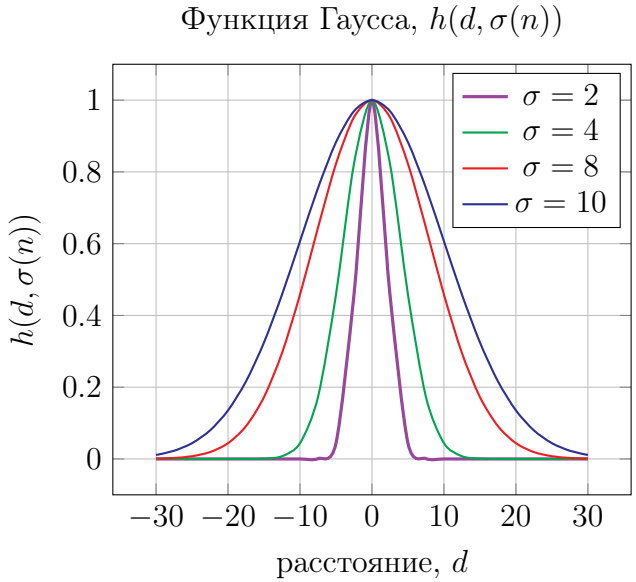


Рис. 3.2: Зависимость функции Гаусса от эффективной ширины

уменьшается при удалении от своего центра. Обозначим символом $h_{j,i}$, топологическую окрестность с центром в победившем нейроне i, состоящую из множества возбуждаемых нейронов, каждый из которых имеет некоторый индекс j. Пусть $d_{j,i}$ расстояние между победившим нейроном i и вторично возбужденным нейроном j, определим значение этого расстояния чуть-чуть позже. Тогда можно предположить, что топологическая окрестность $h_{j,i}$ удовлетворяет следующим требованиям.

- Топологическая окрестность $h_{j,i}$ является симметричной относительно точки максимума, определяемой при $d_{j,i} = 0$ (максимум функции достигается в победившем нейроне, т.е. когда $i = j$).
- Амплитуда топологической окрестности $h_{j,i}$ монотонно уменьшается с увеличением расстояния $d_{j,i}$, достигая нуля при $d_{j,i} \to \infty$. Это необходимое условие сходимости.

Типичным примером такой функции служит функция Гаусса:

$$h_{j,i(\boldsymbol{x})} = \exp\left(-\frac{d^2_{j,i(\boldsymbol{x})}}{2\sigma^2}\right). \tag{3.4}$$

Это выражение является инвариантным к расположению победившего нейрона (т.е. не зависит от его расположения). Параметр σ называется *эффективной шириной* топологической окрестности. Этот параметр определяет уровень, до которого нейроны из топологической окрестности победившего участвуют в процессе обучения (см. рис. 3.2).

Для обеспечения кооперации между соседними нейронами необходимо, чтобы топологическая окрестность $h_{j,i}$ была зависимой от латерального расстояния между победившим i и возбуждаемым j нейроном в выходном пространстве. Именно это свойство отражено в выражении (3.4). В случае одномерной решетки расстояние $d_{j,i}$ является целым числом равным модулю $|j - i|$. В случае двумерной решетки это расстояние определяется соотношением

$$d^2_{j,i} = \|\boldsymbol{r}_j - \boldsymbol{r}_i\|^2, \tag{3.5}$$

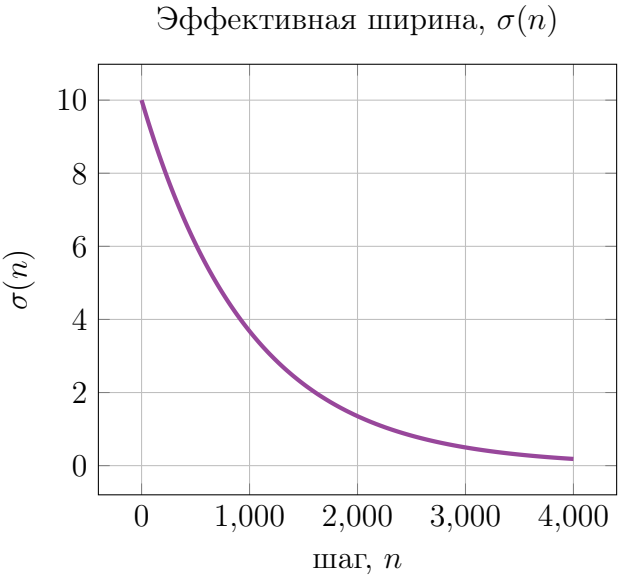


Рис. 3.3: Пример изменения функции эффективной ширины с течением времени

где дискретный вектор $\boldsymbol{r}_j$ определяет позицию возбуждаемого нейрона, а $\boldsymbol{r}_i$ - победившего нейрона. Оба этих измерения проводятся в дискретном выходном пространстве.

Еще одним уникальным свойством алгоритма обучения самоорганизующейся карты является то, что размер топологической окрестности со временем уменьшается. Это требование удовлетворяется за счет постепенного уменьшения ширины σ. Например, можно уменьшать ее экспоненциально в зависимости от дискретного времени n (см. рис. 3.3):

$$\sigma(n) = \sigma_0 \exp\left(-\frac{n}{\tau_1}\right), \; n \in \mathbb{N}, \tag{3.6}$$

где σ_0 - начальное значение величины σ; τ_1 - некоторая константа.

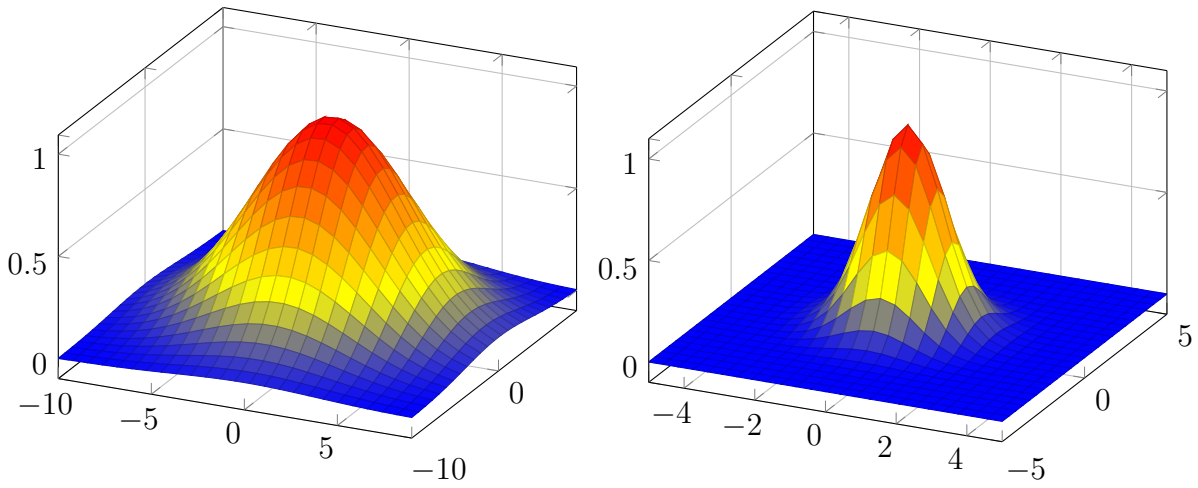
По окончании этапа обучения функция $h_{j,i}$ должна охватывать только ближайших соседей. Исходя из этого мы получаем следующую форму функции топологической окрестности:

$$h_{j,i(\boldsymbol{x})}(n) = \exp\left(-\frac{d_{j,i}^2}{2\sigma^2(n)}\right), \; n \in \mathbb{N}. \tag{3.7}$$

Таким образом, при увеличении количества итераций n ширина $\sigma(n)$ экспоненциально убывает, а вместе с ней соответственно сжимается и топологическая окрестность см. рис. 3.4.

Процесс адаптации Процесс синаптической адаптации включает в себя изменение синаптических весов сети. Для того, чтобы сеть могла самоорганизоваться, вектор синаптических весов $\boldsymbol{w}_j$ нейрона j должен изменяться в соответствии с входным вектором $\boldsymbol{x}$. Заметим, что изменения в связях происходит только в одном направлении, что в конечном счете приводит все веса к точке насыщения. Для того, чтобы обойти эту проблему введем функцию забывания $g(y_i)\boldsymbol{w}_j$. Единственным требование, налагаемым на эту функцию станет равенство

$$g(y_i) = 0, \text{ при } y_i = 0. \tag{3.8}$$



(a) До обучения

(b) После обучения

Рис. 3.4: Изменение топологической окрестности нейрона до и после обучения сети

Имея такую функцию, изменение вектора весов нейрона j в решетке можно выразить следующим образом:

$$\Delta \boldsymbol{w}_j = \eta y_i \boldsymbol{x} - g(y_i)\boldsymbol{w}_j, \tag{3.9}$$

где, η - параметр скорости обучения алгоритма. Для удовлетворения условия (3.8) выберем следующую линейную функцию $g(y_i)$:

$$g(y_i) = \eta y_i. \tag{3.10}$$

Теперь для упрощения выражения (3.9) примем $y_j = h_{j,i(\boldsymbol{x})}$ и получим формулу изменения матрицы весов:

$$\Delta \boldsymbol{w}_j = \eta h_{j,i(\boldsymbol{x})} \cdot (\boldsymbol{x} - \boldsymbol{w}_j); \tag{3.11}$$

$$\boldsymbol{w}_j(n+1) = \boldsymbol{w}_j(n) + \eta(n) h_{j,i(\boldsymbol{x})}(n)(\boldsymbol{x} - \boldsymbol{w}_j(n)). \tag{3.12}$$

Это выражение применяется ко всем нейронам решетки, которые лежат в топологической окрестности победившего нейрона i.

Таким образом, представленный алгоритм ведет к топологическому упорядочиванию пространства признаков во входном пространстве.

Глава 4

Реализация

Перейдем к реализации самоорганизующейся карты Кохонена. Для начала реализуем структуру, которая будет представлять вычислительный нейрон. Как обсуждалось ранее в разделе Модели нейронов общем случае вычислительный нейрон описывается набором синаптических связей, сумматором и функцией активации. Но поскольку нам нужно описать нейрон в модели самоорганизующейся карты, то для описания нейрона потребуется только синаптические связи и его координаты на *карте свойств* (выходной слой). Кроме того дальше нам необходимо вычислять расстояние между нейронами в том пространстве, где они находятся, конкретно, в этом пространстве будет определена евклидова норма, порождающая в нем евклидову метрику. Кроме того нет смысла усложнять описание этой структуры приватными полями и структурами, поэтому достаточно будет реализовать именно `struct neuron`. Итак необходимый нам вычислительный нейрон может быть представлен структурой описанной в листинге 4.1.

```
1  struct neuron {
2      neuron (int x, int y, std::vector<double>&& ws) :
3          coords({x,y}),
4          weights(std::move(ws))
5      {}
6
7      static double distance (const neuron& lhs, const neuron& rhs) {
8          double sq_dist = 0;
9          sq_dist = (rhs.coords.first - lhs.coords.first) *
10                   (rhs.coords.first - lhs.coords.first)
11                                     +
12                   (rhs.coords.second - lhs.coords.second) *
13                   (rhs.coords.second - lhs.coords.second);
14
15         return std::sqrt(sq_dist);
16     }
17
18
19     const std::pair<int, int> coords;
20     std::vector<double> weights;
21 }
```

Листинг 4.1: Реализация вычислительного нейрона карты

Дальше нам понадобиться определять равные между собой нейроны и определять меньший из них, поэтому так же реализуем нужные бинарные операции сравнения см. листинг 4.2.

```
inline bool operator== (const neuron& lhs, const neuron& rhs) {
    return (lhs.coords == rhs.coords);
}

inline bool operator< (const neuron& lhs, const neuron& rhs) {
    const auto& [x1,y1] = lhs.coords;
    const auto& [x2,y2] = rhs.coords;

    return (x1 == x2 ? y1 < y2 : x1 < x2);
}
```

Листинг 4.2: Бинарные операций сравнения **struct neuron**

Начнем реализовывать структуру для описания самой карты. Поскольку к каждому типу нейронной сети необходим индивидуальный подход нет смысла создавать лишние абстракции и использовать ненужные здесь механизмы наследования. Давайте начнем реализовать наш `class sokm` см. листинг 4.3.

```
class sokm {
public:
    sokm ( unsigned int idim,
           unsigned int fdim,
           unsigned int n, unsigned int m ):
        input_dim(idim),
        feature_dim(fdim),
        ewidth0( ((double)std::max(n,m)) / 2 ),
        step(0)
    {
        error_handler::_VERIFY(n*m == fdim, "incorrect feature dimension");

        update_ewidth();
        update_lrate();
        construct_feature_layer(n, m);
    }

private:
    void construct_feature_layer (int n, int m) {
        error_handler::_VERIFY((n*m) != 0 ,
                               "output layer shoud contain at least one dimension");

```

```
23          for (unsigned x = 0; x < n; ++x) {
24              for (unsigned y = 0; y < m; ++y) {
25                  neurons.push_back(
26                          neuron( x, y,
27                              construct_neuron_weights())
28                  );
29              }
30          }
31      }
32
33      std::vector<double> construct_neuron_weights() {
34          std::vector<double> weights(input_dim, 0);
35
36          for (auto& w : weights) {
37              w = rndm::random<double>(-1.0, 1.0);
38          }
39
40          return weights;
41      }
42
43
44      const unsigned int input_dim;
45      const unsigned int feature_dim;
46      const double       ewidth0;
47            unsigned int step;
48            unsigned int epoch = 1;
49      const double       lrate0 = 0.3;
50      const double       tau1 = (1000 / std::log10(ewidth0));
51      const double       tau2 = 1000;
52            double       lrate  = 0;
53            double       ewidth = 0;
54            double       sq_ewidth = ewidth * ewidth;
55      std::vector<neuron> neurons;
56  };
```

Листинг 4.3: Код части реализации **class sokm**

Обновление введенных констант будет происходить с помощью функций представленные в листинге 4.4.

```
1   public:
2       void update_step() {
3           ++step;
4           if (step % 500 == 0) { std::cout <<
5               std::format("epoch: {} step: {}\n", epoch, step); }
6       }
7
8       void update_epoch() {
```

```
9           ++epoch;
10      }
11
12      void update_ewidth() {
13          ewidth = ( step > 1000 ?
14                      ewidth0 * std::exp(-( (step) / (tau1) )) :
15                      ewidth0 );
16          sq_ewidth = ewidth * ewidth;
17      }
18
19      void update_lrate() { // [WARNING]: > for doubles
20          lrate = lrate0 * std::exp(-( (step) / (tau2) ));
21      }
```

Листинг 4.4: Код реализующий обновление констант

Мы обсудили, что процесс обучения нейронной сети сводится к основным модулям процессу адаптации, процессу кооперации и процессу адаптации каждый из них может быть реализован по отдельности как показано в листинге 4.5.

```
1   // find neuron-winner ix (min euclidean distance)
2   // for current input signal
3   const neuron& competition (
4           const std::vector<double>& signal
5       ) const {
6       unsigned ix = neurons.size();
7       double min_sq_dist = std::numeric_limits<double>::max();
8
9       for (unsigned neuron_no {0}; neuron_no < neurons.size(); ++neuron_no) {
10          double dist = sq_euclidean_distance(signal, neurons[neuron_no].weights);
11          if (math::is_double_grt(min_sq_dist, dist)) {
12              min_sq_dist = dist;
13              ix = neuron_no;
14          }
15      }
16
17      return neurons[ix];
18  };
19
20  // find topological neighbourhood
21  // for neuron-winner (nw)
22  std::vector<neuron*> cooperation (const neuron& nw) {
23      std::vector<neuron*> tpn {};
24      double sq_e_width = ewidth * ewidth;
25                                      // cooperation process
26      for (unsigned i {0}; i < neurons.size(); ++i) {
27          double dist = neuron::distance(nw, neurons[i]);
28          double hjix = std::exp(- ((dist * dist) / (2 * sq_e_width)) );
```

```
        if (math::is_double_grt(hjix, 0)) {
            tpn.push_back(&neurons[i]);
        }
    }

    return tpn;
};

void adaptation (const vd& sig, const neuron& nw, const std::vector<neuron*>& tpn) {
    double sq_e_width = ewidth * ewidth;
    for (unsigned i {0}; i < tpn.size(); ++i) {
        // dw  = learning_rate * hjx * (sig - neuron.weights)
        double dist = neuron::distance(nw, neurons[i]);
        double hjix = std::exp(- ((dist * dist) / (2 * sq_e_width)) );
        vd dw = lrate * hjix * (sig - tpn[i]->weights);
        tpn[i]->weights += dw;
    }

    update_lrate();
    update_ewidth();
};
```

Листинг 4.5: Раздельная реализации основных этапов обучения сети

Однако в целях улучшения производительности кода мы поместим реализации процессов кооперации и адаптации в одну функцию `train()`, показанную в листинге 4.6. Она же и станет основной функцией обучения нашей самоорганизующейся карты.

```
public:
    void train (const sdt::vector<double>& signal) {
                                        // competition process
        const neuron& neuron_winner = competition(signal);

                                        // cooperation process
        for (auto& neuron : neurons) {
            const auto [dist, hjix] = hji(neuron_winner, neuron);
            if (math::is_double_grt(hjix, 0)) {

                                        // adaptation process (hjix > 0)
                // dw  = learning_rate * hjix * (signal - neuron.weights)
                vd dw = lrate * hjix * (signal - neuron.weights);
                neuron.weights += dw;
            }
        }

        updatae_constants();
    };
```

Листинг 4.6: Реализации функции **train()**

Для вычисления топологической окрестности нейрона использовалась функция `hji()`, а для вычисления квадрата евклидова расстояния между вектор входного сигнала и вектором весом использовалась функция `sq_euclidean_distance()`. Обе эти функции представлены в листинге 4.7.

```
public:
    void train (const sdt::vector<double>& signal) {
                                        // competition process
        const neuron& neuron_winner = competition(signal);

                                        // cooperation process
        for (auto& neuron : neurons) {
            const auto [dist, hjix] = hji(neuron_winner, neuron);
            if (math::is_double_grt(hjix, 0)) {

                                        // adaptation process (hjix > 0)
                // dw  = learning_rate * hjix * (signal - neuron.weights)
                vd dw = lrate * hjix * (signal - neuron.weights);
                neuron.weights += dw;
            }
        }

        updatae_constants();
    };
```

Листинг 4.7: Реализации функций **hji()** и **sq_euclidean_distance()**

Реализации чтения данных для обучения и места входа программы `main()` представлены в листингах 6.2 и 6.3 в разделе Приложение.

Глава 5

Заключение

5.1 Работа нейронной сети

Мы разобрали теоретическую часть работы нейронной сети Кохонена и реализовали эту модель в виде программного кода. Пришло время проверить ее в действии. Нашей первоначальной задачей была классификация цифр. Для обучения и проверки нашей нейронной сети мы будем пользоваться набором данным MNIST [6]. Этот набор состоит из отцифрованных изображений рукописных цифр от 0 до 9 см. рис. 5.1 и отметок - самих цифр, необходимых, например, для проверки или для обучения с учителем.

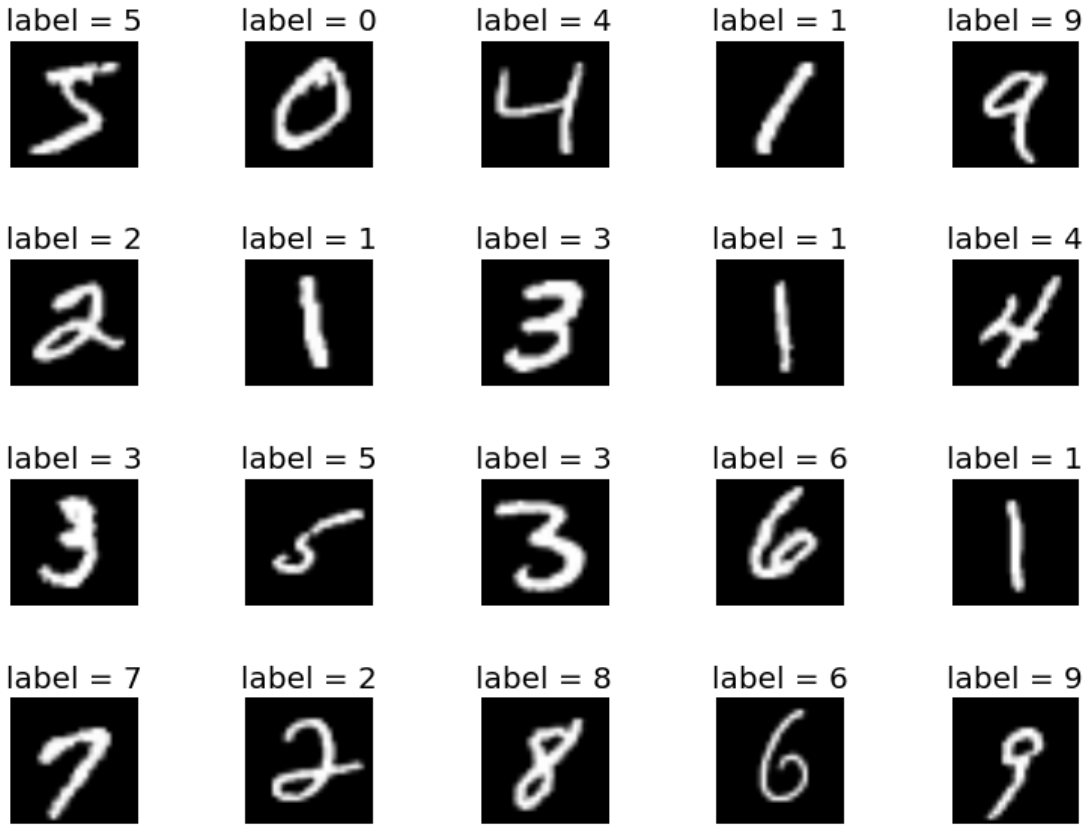


Рис. 5.1: Элементы набора данных **MNIST**

Каждое изображение имеет размер 28 на 28 пикселей, мы будет обучать нашу нейронную сеть с помощью векторов сигналов из 784 элементов, каждому элементу будет соответствовать пиксель, если пиксель черный, то элемент будет равен 0, если же пиксель белый или имеет оттенок серого цвета, то соответствующий элемент входного вектора сигнала будет иметь значение 255. Набор для обучения нейронной сети может включать до 60000 входных сигналов, а для проверки до 10000. Для начала давайте проверим, как работает наша необученная нейронная сеть на тестовом наборе. Итоги работы нейронной сети приведены в таблице 5.1. Изображения делятся на классы по по изображенной на них цифре. Наша нейронная сеть сопоставляет изображение некоторому конкретному классу.

(0,0)	size: 1928	(1,0)	size: 451
[0]: 310	[9]: 167	[0]: 88	[9]: 58
[1]: 25	[8]: 153	[1]: 46	[8]: 16
[2]: 94	[7]: 73	[2]: 45	[7]: 28
[3]: 34	[6]: 590	[3]: 10	[6]: 62
[4]: 398	[5]: 84	[4]: 79	[5]: 19
(0,1)	size: 153	(1,1)	size: 1571
[0]: 2	[9]: 57	[0]: 265	[9]: 230
[1]: 10	[8]: 4	[1]: 7	[8]: 86
[2]: 26	[7]: 8	[2]: 115	[7]: 368
[3]: 6	[6]: 2	[3]: 282	[6]: 42
[4]: 33	[5]: 5	[4]: 69	[5]: 107
(0,2)	size: 215	(1,2)	size: 887
[0]: 3	[9]: 17	[0]: 4	[9]: 256
[1]: 31	[8]: 11	[1]: 3	[8]: 64
[2]: 38	[7]: 49	[2]: 10	[7]: 80
[3]: 19	[6]: 1	[3]: 85	[6]: 58
[4]: 25	[5]: 21	[4]: 205	[5]: 122
(0,3)	size: 95	(1,3)	size: 4420
[0]: 0	[9]: 8	[0]: 287	[9]: 212
[1]: 0	[8]: 1	[1]: 942	[8]: 638
[2]: 4	[7]: 39	[2]: 592	[7]: 377
[3]: 8	[6]: 5	[3]: 547	[6]: 170
[4]: 29	[5]: 1	[4]: 126	[5]: 529
(0,4)	size: 198	(1,4)	size: 82
[0]: 18	[9]: 3	[0]: 3	[9]: 1
[1]: 30	[8]: 1	[1]: 41	[8]: 0
[2]: 103	[7]: 5	[2]: 5	[7]: 1
[3]: 0	[6]: 25	[3]: 19	[6]: 3
[4]: 13	[5]: 0	[4]: 5	[5]: 4

Таблица 5.1: Результат работы необученной карты

Из таблицы видно, что нейронная сеть достаточно плохо справилась с задачей классификации. На самом деле смотреть на работу необученной нейронной сети не совсем интересно. Давайте теперь обучим нашу нейронную сеть на 60000 тренировочных примерах в 2 эпохи обучения. Полученные результаты представлены в таблице 5.2.

(0,0)	size: 1371	(1,0)	size: 923
[0]: 2	[9]: 362	[0]: 0	[9]: 74
[1]: 0	[8]: 87	[1]: 564	[8]: 20
[2]: 6	[7]: 528	[2]: 14	[7]: 143
[3]: 12	[6]: 0	[3]: 33	[6]: 2
[4]: 290	[5]: 84	[4]: 37	[5]: 36
(0,1)	size: 1443	(1,1)	size: 870
[0]: 1	[9]: 504	[0]: 2	[9]: 12
[1]: 0	[8]: 53	[1]: 555	[8]: 42
[2]: 22	[7]: 275	[2]: 24	[7]: 55
[3]: 49	[6]: 10	[3]: 3	[6]: 23
[4]: 477	[5]: 52	[4]: 42	[5]: 112
(0,2)	size: 637	(1,2)	size: 1032
[0]: 9	[9]: 1	[0]: 8	[9]: 9
[1]: 10	[8]: 51	[1]: 6	[8]: 485
[2]: 188	[7]: 8	[2]: 178	[7]: 8
[3]: 36	[6]: 269	[3]: 184	[6]: 6
[4]: 36	[5]: 29	[4]: 0	[5]: 148
(0,3)	size: 763	(1,3)	size: 1257
[0]: 57	[9]: 2	[0]: 29	[9]: 7
[1]: 0	[8]: 26	[1]: 0	[8]: 181
[2]: 503	[7]: 7	[2]: 21	[7]: 0
[3]: 40	[6]: 99	[3]: 647	[6]: 3
[4]: 6	[5]: 23	[4]: 0	[5]: 369
(0,4)	size: 790	(1,4)	size: 914
[0]: 46	[9]: 30	[0]: 826	[9]: 8
[1]: 0	[8]: 10	[1]: 0	[8]: 19
[2]: 58	[7]: 3	[2]: 18	[7]: 1
[3]: 1	[6]: 536	[3]: 5	[6]: 10
[4]: 94	[5]: 12	[4]: 0	[5]: 27

Таблица 5.2: Результат работы обученной карты

Как мы видим из результатов, то обученная самоорганизующаяся карты справляется с задачей классификации намного лучше. Однако стоит отметить важные моменты. У нашей карты возникают некоторые сложности, например, цифры классов 9, 4, 7, или 6, 2 или 3, 5 сложны для определения см. рис. 5.2. И в тоже время она достаточно хорошо справляется с определением классов 0 и 1, иногда путая 1 и 7.

5.2 Рекомендации по улучшению

Построение и обучение нейронной сети требует творческого подхода. каждая нейронная сеть должна быть настроена на свою конкретную задачу, ведь пытаться обучить ее всему подряд будет неэффективно. И, конечно, для каждой конкретной задачи есть свои издержки и хитрости, которые часто обнаруживаются путем проб и ошибок. В этой работе мы не стремились сделать идеально точную нейронную сеть, нашей задачей было лишь ее построение. Однако, для любопытных читателей, что

Рис. 5.2: Элементы набора данных **MNIST**

хотели бы большей точности и производительности, мы оставим несколько идей, которые могли бы им помочь.

- Прежде всего советуем начать экспериментировать с начальными значениями введенных нами констант. Они оказываются важными в начале обучения сети, когда синаптические веса еще не скорректированы, но теряют свое влияние по мере обучения сети.

- Попробуйте сменить способ обучения сети. На наш взгляд, это важнейший процесс, определяющий всю дальнейшую работу нейронной сети. В нашем случае мы использовали конкурентное обучение без учителя, однако, кроме него есть еще обучение с учителем или с подкреплением и каждый со своими подтипами.

- Не стоит ограничиваться самоорганизующимися картами для задачи классификации. Возможно, вам стоит обратить внимание на другие более мощные нейронные сети, например, на CNN (*Convolutional Neural Network*), которые также применяются для задач классификации.

5.3 Вывод

Самоорганизующиеся карты, которые не всегда линейны, имеют преимущество в сохранении структурной информации из обучающих данных в неизменном виде. Другой вид нейронных сетей, такой как "Метод Главных Компонент"может привести к потере информации при использовании, когда сами данные имеют большую размерность, а мы хотим ее уменьшить. Самоорганизующиеся карты могут быть отличной альтернативой МГК для уменьшения размерности, особенно если данные имеют несколько измерений и каждое заданное измерение имеет значение.

Глава 6

Приложение

```cpp
#ifndef UTILS_HPP
#define UTILS_HPP


#include <iostream>
#include <fstream>
#include <cmath>
#include <utility>
#include <map>
#include <set>
#include <vector>
#include <queue>
#include <utility>
#include <string>
#include <random>
#include <format>
#include <algorithm>
#include <functional>


namespace alias {
    using ui = unsigned int;

    using vvd = std::vector<std::vector<double>>;
    using vd  = std::vector<double>;
    using vvi = std::vector<std::vector<int>>;
    using vi  = std::vector<int>;

    using iipair = std::pair<int,int>;
}


namespace rndm {

    template <class Num_t, typename Gen_t = std::mt19937>
```

```
36      Num_t random (const Num_t lower_bound, const Num_t upper_bound) {
37          static Gen_t generator(std::random_device{}());
38
39          using unfdist_type = typename std::conditional <
40              std::is_integral<Num_t>::value,
41              std::uniform_int_distribution<Num_t>,
42              std::uniform_real_distribution<Num_t>
43          >::type;
44
45          static unfdist_type unfdist;
46          return unfdist(
47                  generator,
48                  typename unfdist_type::param_type{lower_bound, upper_bound});
49      }
50
51  }
52
53
54  namespace error_handler {
55      inline void _VERIFY (bool expression, const char* msg) {
56          if (expression == false)
57          {
58              std::cerr << "\n" << "====================================\n"
59                        << msg
60                        << "\n" << "====================================";
61              std::exit(134);
62          }
63      }
64  }
65
66
67  namespace math {
68      static const int32_t kMaxUlps = 4;
69
70      inline bool almost_equal (double x, double y, int32_t maxUlps = kMaxUlps) {
71          error_handler::_VERIFY(maxUlps > 0 && maxUlps < 4 * 1024 * 1024,
72                  "almost_equal: invalid maxUlps");
73
74          int aInt = *(int*)&x;
75          if (aInt < 0) // Make aInt lexicographically ordered as
76                        // a twos-complement int
77              aInt = 0x80000000 - aInt;
78
79          int bInt = *(int*)&y;
80          if (bInt < 0) // Make bInt lexicographically ordered as
81                        // a twos-complement int
82              bInt = 0x80000000 - bInt;
83
```

```
84          int intDiff = abs(aInt - bInt);
85          if (intDiff <= maxUlps)
86              return true;
87
88          return false;
89      }
90
91      inline bool is_double_grt (double x, double y) {
92          return ( x > y + std::numeric_limits<double>::epsilon() );
93      }
94  }
95
96
97  template <class Num_t = double>
98  inline std::vector<Num_t> operator+ ( const std::vector<Num_t>& rhs,
99                                        const std::vector<Num_t>& lhs ) {
100     error_handler::_VERIFY(rhs.size() == lhs.size(),
101                            "vectors shouls have the same size");
102
103     std::vector<Num_t> result (rhs.size());
104     for (unsigned int i {0}; i < rhs.size(); ++i) {
105         result[i] = rhs[i] + lhs[i];
106     }
107
108     return result;
109 }
110
111
112 template <class Num_t = double>
113 inline std::vector<Num_t> operator- ( const std::vector<Num_t>& rhs,
114                                       const std::vector<Num_t>& lhs ) {
115     error_handler::_VERIFY(rhs.size() == lhs.size(),
116                            "vectors shouls have the same size");
117
118     std::vector<Num_t> result (rhs.size());
119     for (unsigned int i {0}; i < rhs.size(); ++i) {
120         result[i] = rhs[i] - lhs[i];
121     }
122
123     return result;
124 }
125
126 template <class Num_t = double>
127 inline std::vector<Num_t>& operator+= ( std::vector<Num_t>& lhs,
128                                         const std::vector<Num_t>& rhs ) {
129     error_handler::_VERIFY(rhs.size() == lhs.size(),
130                            "vectors shouls have the same size");
131     for (unsigned int i {0}; i < lhs.size(); ++i) {
```

```
132         lhs[i] += rhs[i];
133     }
134 
135     return lhs;
136 }
137 
138 
139 template <class Num_t = double>
140 inline std::vector<Num_t> operator* ( const Num_t scal,
141                                       const std::vector<Num_t> vec ) {
142     std::vector<Num_t> result (vec.size());
143     for (unsigned int i = 0; i < vec.size(); ++i) {
144         result[i] = scal * vec[i];
145     }
146 
147     return result;
148 }
149 
150 
151 namespace ccout {
152     inline void print (const std::vector<char>& chars) {
153         std::cout << "{ ";
154         for (const auto& c : chars) {
155             std::cout << (int)c << " ";
156         }
157         std::cout << "}\n";
158     }
159 
160     inline void print (const std::vector<double>& doubles) {
161         std::cout << "{ ";
162         for (const auto& d : doubles) {
163             std::cout << d << " ";
164         }
165         std::cout << "}\n";
166     }
167 
168     inline void print (const alias::iipair& pair) {
169         std::cout << std::format("({},{}) ", pair.first, pair.second);
170     }
171 }
172 
173 
174 namespace vec_utils {
175     template <class Valty>
176     inline std::vector<double> normalize_vector (
177             const std::vector<Valty>& vals
178     ) {
179         std::vector<double> result(vals.size());
```

```
180         for (unsigned int i = 0; i < vals.size(); ++i) {
181             // result[i] = (double)vals[i];
182             result[i] = ((int)vals[i] == 0 ? 0.0 : 255.0);
183         }
184 
185         return result;
186     }
187 }
188 
189 
190 #endif // UTILS_HPP
```

Листинг 6.1: Полный код реализации вспомогательных пространств имен

```
1  #ifndef SOKM_HPP
2  #define SOKM_HPP
3  
4  
5  #include "utils.hpp"
6  #include "read_mnist.hpp"
7  
8  #define DEBUG_PRINT_TRAIN 0
9  #define DEBUG_PRINT_STEP 0
10 
11 struct neuron {
12     using pair = alias::iipair;
13     using vd   = alias::vd;
14 
15 
16     neuron (int x, int y, vd&& ws) : coords({x,y}), weights(std::move(ws)) {}
17 
18     static double distance (const neuron& lhs, const neuron& rhs) {
19         double sq_dist = 0;
20         sq_dist = (rhs.coords.first - lhs.coords.first) *
21                   (rhs.coords.first - lhs.coords.first)
22                                     +
23                   (rhs.coords.second - lhs.coords.second) *
24                   (rhs.coords.second - lhs.coords.second);
25 
26         return std::sqrt(sq_dist);
27     }
28 
29 
30     const pair coords;
31     vd weights;
32 };
33 
```

```
34  inline bool operator== (const neuron& lhs, const neuron& rhs) {
35      return (lhs.coords == rhs.coords);
36  }
37
38  inline bool operator< (const neuron& lhs, const neuron& rhs) {
39      const auto& [x1,y1] = lhs.coords;
40      const auto& [x2,y2] = rhs.coords;
41
42      return (x1 == x2 ? y1 < y2 : x1 < x2);
43  }
44
45
46  class sokm {
47      using ui = unsigned int;
48      using vvd = alias::vvd;
49      using vd = alias::vd;
50      using vi = alias::vi;
51
52  public:
53      sokm ( ui idim,
54             ui fdim,
55             ui n, ui m ):
56          input_dim(idim),
57          feature_dim(fdim),
58          ewidth0( ((double)std::max(n,m)) / 2 ),
59          step(0)
60      {
61          error_handler::_VERIFY(n*m == fdim, "incorrect feature dimension");
62
63          update_ewidth();
64          update_lrate();
65          construct_feature_layer(n, m);
66      }
67
68  private:
69      // construct output neuron layer
70      // [NOTE]: n*m shouldn't equals 0
71      // ^
72      // |
73      // | m
74      // |     n
75      // + -------->
76      void construct_feature_layer (int n, int m) {
77          error_handler::_VERIFY((n*m) != 0,
78              "output layer shoud contain at least one dimension");
79
80          for (ui x = 0; x < n; ++x) {
81              for (ui y = 0; y < m; ++y) {
```

```
                neurons.push_back(
                        neuron( x,y,
                            construct_neuron_weights())
                );
            }
        }
    }

    vd construct_neuron_weights() {
        vd weights(input_dim, 0);

        for (auto& w : weights) {
            w = rndm::random<double>(-1.0, 1.0);
        }

        return weights;
    }

public:
    // square of euclidean distance
    static double sq_euclidean_distance (const vd& x, const vd& y) {
        error_handler::_VERIFY(x.size() == y.size(),
                               "vectors should have same dimensions");
        double sq_dist = 0;

        for (ui i {0}; i < x.size(); ++i) {
            sq_dist += (x[i] - y[i]) * (x[i] - y[i]);
        }

        return sq_dist;
    }

private:
    // find neuron-winner ix (min euclidean distance)
    // for current input signal
    const neuron& competition (const vd& signal) const {
        ui ix = neurons.size(); // no of neuron-winner
        double min_sq_dist = std::numeric_limits<double>::max();

                                        // competition process
        for (ui neuron_no {0}; neuron_no < neurons.size(); ++neuron_no) {
            double dist = sq_euclidean_distance(signal,
                                neurons[neuron_no].weights);
            if (math::is_double_grt(min_sq_dist, dist)) {
                min_sq_dist = dist;
                ix = neuron_no;
            }
        }
```

```
130
131             return neurons[ix];
132         };
133
134         // find topological neighbourhood
135         // for neuron-winner (nw)
136         std::vector<neuron*> cooperation (const neuron& nw) {
137             // topological neighbourhood h_{j,i(x)}
138             std::vector<neuron*> tpn {};
139             double sq_e_width = ewidth * ewidth;
140                                             // cooperation process
141             for (ui i {0}; i < neurons.size(); ++i) {
142                 double dist = neuron::distance(nw, neurons[i]);
143                 double hjix = std::exp(- ((dist * dist) / (2 * sq_e_width)) );
144                 if (math::is_double_grt(hjix, 0)) {
145                     tpn.push_back(&neurons[i]);
146                 }
147             }
148
149             return tpn;
150         };
151
152         void adaptation (
153                 const vd& sig, const neuron& nw,
154                 const std::vector<neuron*>& tpn
155         ) {
156             double sq_e_width = ewidth * ewidth;
157                                               // adaptation process
158             for (ui i {0}; i < tpn.size(); ++i) {
159                 // dw  = learning_rate * hjx * (sig - neuron.weights)
160                 double dist = neuron::distance(nw, neurons[i]);
161                 double hjix = std::exp(- ((dist * dist) / (2 * sq_e_width)) );
162                 vd dw = lrate * hjix * (sig - tpn[i]->weights);
163                 tpn[i]->weights += dw;
164             }
165
166             update_lrate();
167             update_ewidth();
168         };
169
170     public:
171         void train (const vd& signal) {
172             if (step == 0) { printf("step: %u\n", step); }
173
174             const neuron& neuron_winner = competition(signal);
175
176             // double sq_ewidth = ewidth * ewidth;
177                                             // cooperation process
```

```
178         for (auto& neuron : neurons) {
179             const auto [dist, hjix] = hji(neuron_winner, neuron);
180             if (math::is_double_grt(hjix, 0)) {
181                                         // adaptation process (hjix > 0)
182                 // dw  = learning_rate * hjix * (signal - neuron.weights)
183                 vd dw = lrate * hjix * (signal - neuron.weights);
184                 neuron.weights += dw;
185             }
186         }
187 
188         updatae_constants();
189 // ====================================================
190 #if DEBUG_PRINT_TRAIN
191         using std::cout;
192         using std::right;
193 
194         printf("{\n");
195         for (const auto& n : neurons) {
196             printf("\t(%d,%d): { ", n.coords.first, n.coords.second);
197             for (const auto& w : n.weights) {
198                 printf("%f ", w);
199             }
200             printf("};\n");
201         }
202 
203         cout.width(50); cout << right << "===========\n";
204         cout.width(50); cout << right <<
205             std::format("lrate: {}\n", lrate);
206         cout.width(50); cout << right <<
207             std::format("ewidth: {}\n", ewidth);
208 
209         printf("}\n");
210 #endif
211 // ====================================================
212     };
213 
214 private:
215     void updatae_constants() {
216         update_step();
217         update_lrate();
218         update_ewidth();
219     }
220 
221     std::pair<double, double> hji (const neuron& n1, const neuron& n2) {
222         double dist = neuron::distance(n1, n2);
223         return {dist, std::exp(- ((dist * dist) / (2 * sq_ewidth)) )};
224     }
225 
```

```
226     void update_step() {
227         ++step;
228         if (step % 500 == 0) { std::cout <<
229             std::format("epoch: {} step: {}\n", epoch, step); }
230 #if DEBUG_PRINT_STEP
231         std::cout << std::format("lrate: {}\n", lrate);
232         std::cout << std::format("ewidth: {}\n", ewidth);
233 #endif
234     }
235 
236     void update_epoch() {
237         ++epoch;
238     }
239 
240     void update_ewidth() {
241         ewidth = ( step > 1000 ?
242                     ewidth0 * std::exp(-( (step) / (tau1) )) :
243                     ewidth0 );
244         sq_ewidth = ewidth * ewidth;
245     }
246 
247     void update_lrate() { // [WARNING]: > for doubles
248         // lrate = ( math::is_double_grt(lrate0,lrate) ?
249         //           lrate0 :
250         //           lrate0 * std::exp(-( (step) / (tau2) )) );
251         lrate = lrate0 * std::exp(-( (step) / (tau2) ));
252     }
253 
254 public:
255     std::pair<int, int> classify (const vd& signal) const {
256         return competition(signal).coords;
257     }
258 
259     friend void sokm_education_mnist (sokm& map,
260                                       std::string file_path,
261                                       alias::ui epochs);
262 
263 private:
264     const ui input_dim;
265     const ui feature_dim;
266     const double ewidth0;       // sigma0
267     ui step;                    // learnin step,n
268     ui epoch = 1;
269     const double lrate0 = 0.3;  // eta0 0.1
270     const double tau1   = (1000 / std::log10(ewidth0)); // ewidth multiplier
271     const double tau2   = 1000; // lrate multiplier
272           double lrate  = 0;    // [TODO]: should be function
273           double ewidth = 0;    // [TODO]: should be function effective width
```

```
274             double sq_ewidth = ewidth * ewidth;
275     std::vector<neuron> neurons; // vector of output(feature)
276                                  // neuron layer
277 };
278
279
280 namespace ccout { // custom console output
281     using std::cout;
282     using std::right;
283
284     inline void print (const neuron& n) {
285         printf("\t(%d,%d): { ", n.coords.first, n.coords.second);
286         for (const auto& w : n.weights) {
287             printf("%f ", w);
288         }
289         printf("};\n");
290     }
291
292     inline void print (const sokm& map) {
293         printf("{\n");
294         for (const auto& n : map.neurons) {
295             print(n);
296         }
297
298         cout.width(50); cout << right << "===========\n";
299         cout.width(50); cout << right <<
300             std::format("lrate: {}\n", map.lrate);
301         cout.width(50); cout << right <<
302             std::format("ewidth: {}\n", map.ewidth);
303
304         printf("}\n");
305     }
306 }
307
308
309 void sokm_education_mnist (
310         sokm& map, std::string file_path,
311         alias::ui epochs = 1
312     ) {
313     std::ifstream file (file_path, std::ios::in | std::ios::binary);
314
315     if (file.is_open()) {
316         uint32_t magic = 0;
317         uint32_t num_items = 0;
318         uint32_t rows = 0;
319         uint32_t cols = 0;
320
321         file.read(reinterpret_cast<char*>(&magic), 4);
```

```
322         magic = mnist::swap_endian(magic);
323         error_handler::_VERIFY(magic == 2051, "incorrect image file magic");
324 
325         file.read(reinterpret_cast<char*>(&num_items), 4);
326         num_items = mnist::swap_endian(num_items);
327 
328         file.read(reinterpret_cast<char*>(&rows), 4);
329         rows = mnist::swap_endian(rows);
330 
331         file.read(reinterpret_cast<char*>(&cols), 4);
332         cols = mnist::swap_endian(cols);
333 
334         std::cout << std::format("Images: {}\n", num_items);
335         std::cout << std::format("\tRows: {}\n", rows);
336         std::cout << std::format("\tCols: {}\n", cols);
337 
338         char* pixels = new char[rows * cols];
339         while (epochs --> 0) {
340             for (int item = 0; item < num_items; ++item) {
341                 file.read(pixels, rows*cols);
342 
343                 std::vector<char> vector_chars (
344                         pixels, pixels + rows*cols
345                 );
346 
347                 map.train(vec_utils::normalize_vector(vector_chars));
348             }
349             map.update_epoch();
350         }
351         delete[] pixels;
352     }
353 }
354 
355 
356 decltype(auto) sokm_check_mnist ( const sokm& map,
357                         std::string images_path,
358                         std::string labels_path ) {
359     using alias::ui;
360 
361     std::ifstream images_file (images_path, std::ios::in | std::ios::binary);
362     std::ifstream labels_file (labels_path, std::ios::in | std::ios::binary);
363 
364     std::map<alias::iipair, alias::vi> marks;
365     for (const auto& n : map.neurons) {
366         marks.emplace( n.coords, alias::vi{} );
367     }
368 
369     if (images_file.is_open()) {
```

```
370         uint32_t magic = 0;
371         uint32_t num_items = 0;
372         uint32_t num_labels = 0;
373         uint32_t rows = 0;
374         uint32_t cols = 0;
375 
376         images_file.read(reinterpret_cast<char*>(&magic), 4);
377         magic = mnist::swap_endian(magic);
378         error_handler::_VERIFY(magic == 2051,
379                 "incorrect image file magic");
380 
381         labels_file.read(reinterpret_cast<char*>(&magic), 4);
382         magic = mnist::swap_endian(magic);
383         error_handler::_VERIFY(magic == 2049,
384                 "incorrect label file magic");
385 
386         images_file.read(reinterpret_cast<char*>(&num_items), 4);
387         num_items = mnist::swap_endian(num_items);
388 
389         labels_file.read(reinterpret_cast<char*>(&num_labels), 4);
390         num_labels = mnist::swap_endian(num_labels);
391         std::cout << std::format("labels: {},items: {}\n",
392                 num_labels, num_items);
393         error_handler::_VERIFY(num_items == num_labels,
394                 "numbers of labels and items doest't match");
395 
396         images_file.read(reinterpret_cast<char*>(&rows), 4);
397         rows = mnist::swap_endian(rows);
398 
399         images_file.read(reinterpret_cast<char*>(&cols), 4);
400         cols = mnist::swap_endian(cols);
401 
402         std::cout << std::format("Images: {}\n", num_items);
403         std::cout << std::format("\tRows: {}\n", rows);
404         std::cout << std::format("\tCols: {}\n", cols);
405 
406         char label_char;
407         char* pixels = new char[rows * cols];
408 
409         for (int item = 0; item < num_items; ++item) {
410             images_file.read(pixels, rows*cols);
411             labels_file.read(&label_char, 1);
412             int label = std::stoi(std::to_string(int(label_char)));
413 
414             std::vector<char> vector_chars (
415                     pixels, pixels + rows*cols
416             );
417 
```

```
418             auto coords = map.classify (
419                     vec_utils::normalize_vector(vector_chars)
420             );
421 
422             marks.at(coords).push_back(label);
423         }
424 
425         delete[] pixels;
426     }
427 
428     return marks;
429 }
430 
431 
432 #endif // SOKM_HPP
```

Листинг 6.2: Полный код реализации **struct neuron** и **class sokm**

```
1  #include "sokm.hpp"
2  #include "read_mnist.hpp"
3  
4  int main() {
5  
6      sokm map (28*28, 10, 2, 5);
7      sokm_education_mnist(map, "../data/train-images-idx3-ubyte", 2);
8      const auto marks = sokm_check_mnist ( map,
9              "../data/t10k-images-idx3-ubyte",
10             "../data/t10k-labels-idx1-ubyte" );
11 
12     using std::cout;
13     using std::right;
14     using std::left;
15     using std::format;
16 
17     std::vector<int> checker(10, 0);
18     alias::ui fsize = 0;
19 
20     for (const auto& [coords, labels] : marks) {
21         cout << format("({},{}) size: {}\n", coords.first,
22                 coords.second, labels.size());
23         for (int l = 0; l < 5; ++l) {
24             auto size1 = count(labels.begin(), labels.end(), l);
25             auto size2 = count(labels.begin(), labels.end(), (9-l));
26 
27             cout << "\t"; cout.width(11);
28             cout << left << format("[{}]: {}", l, size1);
29             cout << left << format("[{}]: {}\n", (9-l), size2);
```

```
30
31              fsize += size1; checker[l] += size1;
32              fsize += size2; checker[9-l] += size2;
33          }
34      }
35
36      cout << format("Total number: {}\n", fsize);
37      for (alias::ui i = 0; i < 5; ++i) {
38          cout << "\t"; cout.width(12);
39          cout << left << format("[{}]: {}", i, checker[i]);
40          cout << format("[{}]: {}\n", (9-i), checker[9-i]);
41      }
42
43      return 0;
44  }
```

Листинг 6.3: Полный код точки входа программы **main()**

Список иллюстраций

Список листингов

Список литературы

[1] Поляков Г. И. О принципах нейронной организации мозга, М: МГУ, 1965

[2] Хайкин С. Нейронные сети: полный курс - Neural Networks: A Comprehensive Foundation. 2-е изд. — М.: Вильямс, 2006. — 1104 с.

[3] Kohonen T. Self-organizing maps. – Springer Science & Business Media, 2012. – Т. 30.

[4] THE MNIST DATABASE : офиц. сайт. - URL: http://yann.lecun.com/exdb/mnist (дата обращения: 29.12.2023).