ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ «САМАРСКИЙ НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ
ИМЕНИ АКАДЕМИКА С.П. КОРОЛЕВА

Лекции по курсу «Интеллектуальные системы»

Для студентов направления 090401

Очное обучение 2023-2024 учебный год

Лектор: Солдатова Ольга Петровна, к.т.н., доцент

Глубокое обучение.

Лекция 6

Глубокие сети: преимущество и сложность.
Регуляризация в нейронных сетях.
Инициализация весов.
Нормализация по мини-батчам.
Нормализация по весам.
Современные функции активации.
Алгоритм градиентного спуска с моментом.
Адаптивные варианты градиентного спуска.
Операция свёртки.
Двумерная свёртка.
Свёрточные сети.
Субдискретизация.
Архитектура сети LeNet-5.

Литература по курсу

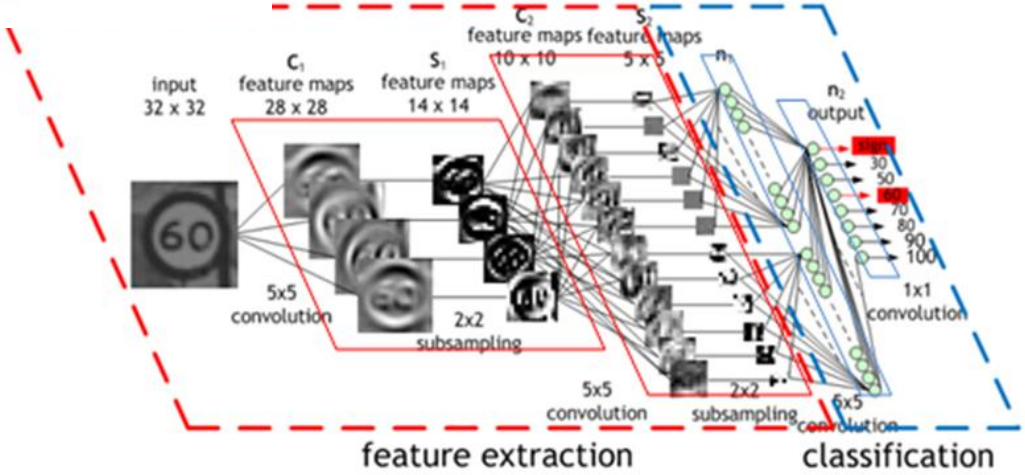
1. Куприянов А.В. «Искусственный интеллект и машинное обучение» https://do.ssau.ru/moodle/course/view.php?id=1459
2. Столбов В.Ю. Интеллектуальные информационные системы управления предприятием / В.Ю. Столбов, А.В. Вожаков, С.А. Федосеев. – Москва : Издательство Литрес, 2021. – 470 с. – Режим доступа: по подписке. – URL: https://www.litres.ru/book/artem-viktorovich-vo/intellektualnye-informacionnye-sistemy-upravleniya-pr-66403444/.
3. Осовский С. Нейронные сети для обработки информации / Пер. с пол. И.Д. Рудинского. – М.: Финансы и статистика, 2002. – 344 с.: ил.
4. Горбаченко, В. И. Интеллектуальные системы: нечеткие системы и сети : учебное пособие для вузов / В. И. Горбаченко, Б. С. Ахметов, О. Ю. Кузнецова. — 2-е изд., испр. и доп. — Москва : Издательство Юрайт, 2018. — 105 с. — (Университеты России). — ISBN 978-5-534-08359-0. — Текст : электронный // ЭБС Юрайт [сайт]. — URL: https://urait.ru/bcode/424887 – Режим доступа: https://urait.ru/bcode/424887
5. 4. Гафаров Ф.М. Искусственные нейронные сети и приложения: учеб. пособие / Ф.М. Гафаров, А.Ф. Галимянов. – Казань: Изд-во Казан. ун-та, 2018. – 121 с. – Текст : электронный. – Режим доступа: https://repository.kpfu.ru/?p_id=187099
6. Хайкин С. Нейронные сети: Полный курс: Пер. с англ. - 2-е изд. – М.: Вильямс, 2006. – 1104 с.: ил.
7. Борисов В.В., Круглов В.В., Федулов А.С. Нечёткие модели и сети. – М.: Горячая линия–Телеком, 2007. -284 с.: ил.
8. Рутковская Д., Пилиньский М., Рутковский Л. Нейронные сети, генетические алгоритмы и нечёткие системы: Пер. с польск. И.Д.Рудинского, - М.: Горячая линия – Телеком, 2007. – 452 с. ил.
9. Николенко С., Кадурин А., Архангельская Е. Глубокое обучение. — СПб.: Питер, 2018. — 480 с.: ил. — (Серия ≪Библиотека программиста≫).
10. Гудфеллоу Я., Бенджио И., Курвилль А. Глубокое обучение / пер. с анг. А. А. Слинкина. – 2-е изд., испр. – М.: ДМК Пресс, 2018. – 652 с.: цв. ил.

Глубокое обучение

Решаемые проблемы

- Переобучение
- Привыкание к данным
- Автоматическое выделение признаков

ИДЕЯ: Выделение признаков + классификация



Глубокие сети: преимущество и сложность

Сейчас для обучения нейронных сетей используются разные варианты **алгоритмов градиентного спуска**, а для подсчёта градиента используются **алгоритмы автоматического дифференцирования на графе вычислений**. Все эти методы, никак не зависят от архитектуры нейронной сети и теоретически должны работать для любого графа вычислений, если реализовывать процедуру обратного распространения ошибки.

Идея графа вычислений не является сложной, о том, как дифференцировать композицию функций, известно на протяжении нескольких веков, алгоритм градиентного спуска был известен даже раньше XX века. Сама идея построить глубокую сеть, то есть нанизать друг на друга несколько уровней нейронов, тоже не кажется сложной. **Однако идея глубокого обучения реализовалась только в 2005-2006 годах.**

На самом деле первые глубокие сети появились еще в середине 1960-х годов: первые настоящие глубокие сети в виде глубоких персептронов были описаны **в работах советского ученого А. Г. Ивахненко.** Метод группового учета аргументов стал его основным результатом, Хотя его работы получили широкое признание во всем мире, сейчас его редко вспоминают среди первооткрывателей глубокого обучения; только немецкий учёный Шмидхубер ссылается на его работы.

Глубокие сети: преимущество и сложность

Метод группового учета аргументов выглядит на удивление современно. Если в нем в качестве базовой модели выбрать персептрон, то получим типичную нейронную сеть с несколькими слоями, которая обучается слой за слоем: сначала первый, потом он фиксируется и начинается обучение второго, и т.д. Уже в начале 1970-х годов этим методом вполне успешно обучались модели вплоть до семи уровней в глубину, и это очень похоже на процедуру предобучения без учителя.

Однако первой глубокой нейронной сетью можно считать **Neocognitron Кунихиро Фукусимы**, в котором появились и сверточные слои, и активации, очень похожие на **ReLU**. Эта модель не обучалась в современном смысле этого слова: веса сети устанавливались из локальных правил обучения без учителя.

Примерно в то же время появились и **глубокие модели на основе обратного распространения.** Первым применением обратного распространения ошибки к произвольным архитектурам можно считать **работы финского студента Сеппо Линненмаа:** в 1970 году он построил правила автоматического дифференцирования по графу вычислений, в том числе и обратное распространение. Однако к нейронным сетям эти идеи были применены в работах Дрейфуса и Вербоса, а начиная с классических **работ Румельхарта, Хинтона и Уильямса 1986 года**, метод обратного распространения ошибки стал общепринятым для обучения любых нейронных сетей.

Глубокие сети: преимущество и сложность

Зачем вообще нужны глубокие сети? Теорема Хорника, основанная на более ранних работах Колмогорова, утверждает, что любую непрерывную функцию можно сколь угодно точно приблизить нейронной сетью с одним скрытым слоем.

Оказывается, глубокие архитектуры часто позволяют приблизить одни и те же функции гораздо более эффективно, чем неглубокие. Существуют аналогичные утверждения о булевых схемах: известно, что схемы глубиной к + 1 могут эффективно выразить строго больше булевых функций, чем схемы глубиной к.

Со слоями нейронной сети возникает примерно тот же эффект: одну и ту же функцию часто можно гораздо лучше приблизить более глубокой сетью, чем мелкой, даже если общее число нейронов в сети оставить постоянным. А другая сторона преимущества глубоких сетей —это то, что глубокая нейронная сеть создает еще и распределенное представление.

Почему глубокие сети долго не использовались?

На этот вопрос есть два ответа. Первый — математический. Дело в том, что алгоритм градиентного спуска для глубоких сетей без дополнительных модификаций работать не будет. Если обучать глубокую сеть алгоритмом обратного распространения ошибки, то последний, ближайший к выходам уровень обучится быстро и очень хорошо.

Глубокие сети: преимущество и сложность

Дальше окажется, что большинство нейронов последнего уровня на всех тестовых примерах уже «определились» со своим значением, то есть их выход близок или к нулю, или к единице. Если у них классическая сигмоидальная функция активации, то это значит, что производная у этой функции активации близка к нулю с обеих сторон, однако на неё надо умножить все градиенты в алгоритме обратного распространения.

Получается, что обучившийся последний слой нейронов «блокирует» распространение градиентов назад по графу вычислений, и более ранние уровни глубокой сети в результате фактически не обучаются. Этот эффект называется **проблемой затухающих градиентов (vanishing gradients).**

В **рекуррентных сетях**, которые по определению являются очень глубокими, возникает еще и обратная проблема: иногда **градиенты могут начать «взрываться»**, экспоненциально увеличиваться по мере «разворачивания» нейронной сети (exploding gradients). Обе проблемы возникают часто, и достаточно долго их не могли их удовлетворительно решить.

Глубокие сети: преимущество и сложность

В середине 2000-х годов появились первые действительно хорошо работающие и хорошо масштабирующиеся конструкции. Решение заключалось в том, чтобы предобучать нейронные сети уровень за уровнем с помощью специального случая ненаправленной графической модели, так называемой ограниченной машины Больцмана (restricted Boltzmann machine, RBM).

Получается, что просто так глубокие сети обучать не удается, и нужно сначала последовательно обучать отдельные слои сети совершенно другими алгоритмами, а потом только локально ≪докручивая≫ их градиентным спуском. Однако в настоящее время машины Больцмана практически не используются. Сейчас библиотеки глубокого обучения просто строят граф вычислений, а затем считают градиенты. Появился ряд важных инструментов, которые можно отнести либо к регуляризации, либо к разным модификациям оптимизации в нейронных сетях.

Второй ответ на вопрос о том, почему глубокие нейронные сети было сложно обучать, технический. Дело в том, что раньше компьютеры были медленнее, а доступных для обучения данных было гораздо меньше, чем сейчас.

Регуляризация в нейронных сетях

Современные нейронные сети — это модели с огромным числом параметров, она может содержать миллионы весов. Модель, у которой слишком много свободных параметров, плохо обобщается. Значит, её надо регуляризовать.

Первая идея: ограничить значения каждой переменной, которую оптимизируем. Это можно сделать, добавив к целевой функции регуляризаторы в любом удобном виде. Обычно используют два их вида:

- **L2 -регуляризатор**, сумма квадратов весов $\lambda \sum_w w^2$;
- **L1 -регуляризатор**, сумма модулей весов $\lambda \sum_w |w|$.

В теории нейронных сетей такая регуляризация называется сокращением весов (weight decay), потому что действительно приводит к уменьшению их абсолютных значений. Этот способ известен ещё с 80-х годов XX века.

В таких библиотеках, как TensorFlow и Keras, можно легко применить сокращение весов. Регуляризация в форме сокращения весов применяется до сих пор, однако сильно увлекаться подбором регуляризаторов не стоит.

Во-первых, есть еще один метод регуляризации: надо выделить часть обучающей выборки (**валидационную выборку**) и при обучении на основной обучающей выборке заодно вычислять ошибку и на валидационной выборке, которая будет хорошо оценивать ошибку и на тестовой выборке, ведь она взята из обучающих данных, на которых сеть не обучалась. Остановить обучение нужно будет, когда начнет ухудшаться ошибка на валидационной выборке.

Регуляризация в нейронных сетях

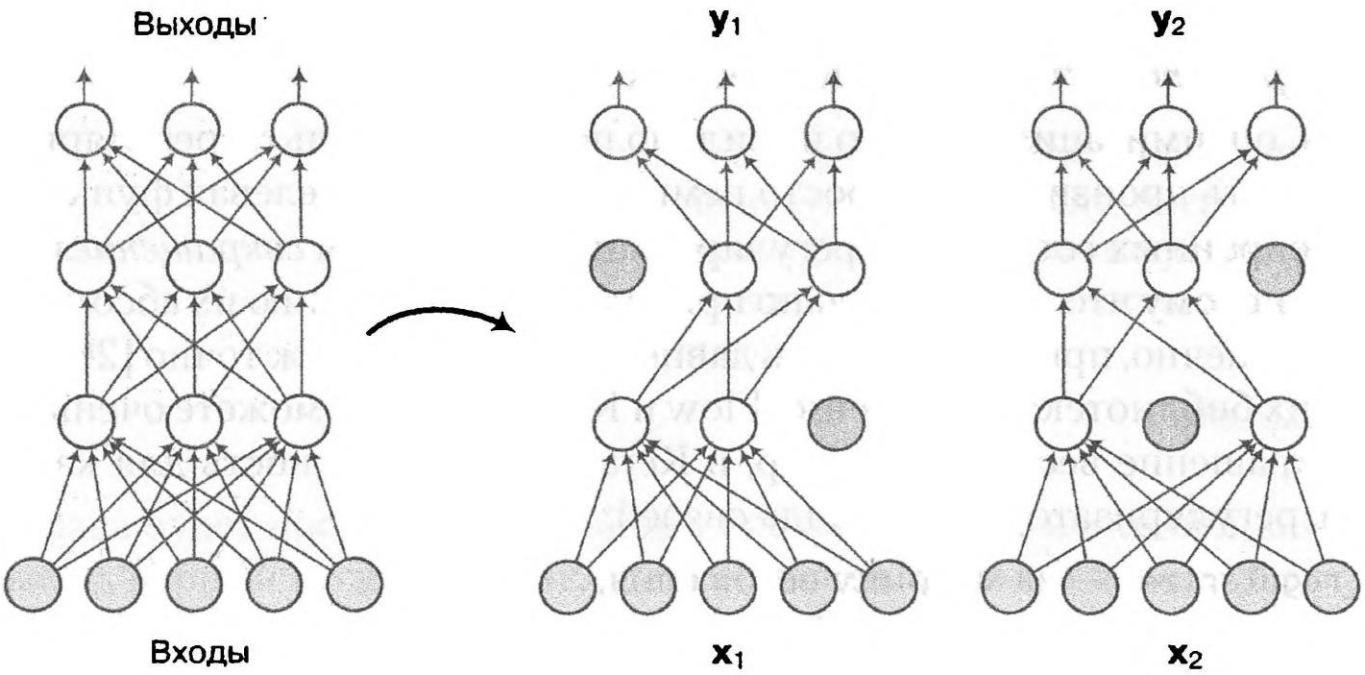
В теории нейронных сетей этот подход обычно называется **методом ранней остановки** (early stopping).

Во-вторых, в последние годы были разработаны более эффективные методы, которые стоит применить в первую очередь. Одним из важнейших методов регуляризации нейронных сетей стал **дропаут**. Пример дропаута показан на рисунке 22, где изображена трехслойная нейронная сеть с пятью входами и тремя слоями по три нейрона.

Для каждого нейрона (кроме нейронов выходного слоя) устанавливается некоторая вероятность p, с которой он будет исключён из сети. Алгоритм обучения меняется таким образом: для каждого нового обучающего примере x каждый нейрон с вероятностью p, либо используется нейрон как обычно, либо его выход устанавливается строго равным нулю. Дальше все происходит без изменений; ноль на выходе приводит к тому, что нейрон фактически выпадает из графа вычислений. На выходном слое дропаут обычно не делают, так как требуется выход определенной размерности, и все его компоненты обычно нужны.

Основной математический результат дропаута состоит в том, как потом применять обученную сеть. На самом деле большой вычислительной сложности здесь нет: усреднение будет эквивалентно применению сети, в которой никакие нейроны не выброшены, но выход каждого нейрона умножен на вероятность p, с которой нейрон оставляли при обучении. Математическое ожидание выхода нейрона при этом сохранится.

Регуляризация в нейронных сетях (рисунок 22)



Инициализация весов

Обучение сети — это большая и сложная задача оптимизации в пространстве очень высокой размерности. Решается она чаще всего методом градиентного спуска — это метод, который ищет только локальный минимум/максимум. Не известно никаких методов глобальной оптимизации, которые позволили бы найти самый лучший локальный оптимум для такой сложной задачи, как обучение глубокой нейронной сети. Поэтому, одним из ключевых вопросов является вопрос, где начинать этот локальный поиск: в зависимости от качества начального приближения решением будут разные локальные оптимумы. Хорошая инициализация весов может позволить обучать глубокие сети и лучше, и быстрее.

Первая идея, которая привела к большим успехам в этом направлении, была в использовании **предобучения без учителя** (unsupervised pretraining). Предобучение широко использовалось в середине 2000-х годов. Сейчас его тоже можно использовать, но это достаточно сложная и дорогостоящая процедура. Получается, что нужно делать две разных процедуры обучения, причем предобучение будет более сложной фазой, чем обучение с учителем. Для того, чтобы обойтись без преобучения, надо понять, почему случайная инициализация работает плохо и что можно сделать для того, чтобы ее улучшить.

Инициализация весов

Ответ был дан в совместной работе Ксавье Глоро (Xavier Glorot) и Йошуа Бенджи, вышедшей в 2010 году. В ней был предложен простой способ инициализации весов, позволяющий существенно ускорить обучение и улучшить качество. Он получил название **инициализации Ксавье** (Xavier initialization).

До данной работы стандартным эвристическим способом случайно инициализировать веса глубокой сети было равномерное распределение следующего вида:

$w_i \sim U\left[-\frac{1}{\sqrt{n_{out}}}, \frac{1}{\sqrt{n_{out}}}\right]$, n_{out} - число выходных нейронов.

В этом случае получается, что дисперсия весов:

$$Var(w_i) = \frac{1}{12}\left(\frac{1}{\sqrt{n_{out}}} + \frac{1}{\sqrt{n_{out}}}\right)^2 = \frac{1}{3n_{out}},$$

и $n_{out} Var(w_i) = \frac{1}{3}$. (223)

После нескольких слоев такое преобразование параметров распределения значений между слоями сети фактически приводит к затуханию сигнала: дисперсия результата слоя каждый раз уменьшается, фактически делится на 3, а среднее у него было нулевое. Аналогичная ситуация повторяется и на шаге обратного распространения ошибки при обучении.

Инициализация весов

Идея авторов статьи заключается в том, что для беспрепятственного распространения значений аргумента функции активации и градиента по сети, дисперсия в обоих случаях должна быть примерно равна единице. Поскольку для неодинаковых размеров слоев невозможно удовлетворить оба условия одновременно, они предложили инициализировать веса очередного слоя сети симметричным распределением с такой дисперсией:

$$Var(w_i) = \frac{2}{n_{in} + n_{out}}$$

что для равномерной инициализации приводит к следующему распределению: $w_i \sim U\left[-\frac{\sqrt{6}}{\sqrt{n_{in}+n_{out}}}, \frac{\sqrt{6}}{\sqrt{n_{in}+n_{out}}}\right]$ (224)

В данной работе был сделан и еще один практически важный вывод. Оказывается, логистическая сигмоида — это весьма неудачная функция активации для глубоких нейронных сетей. Эксперименты с глубокими сетями, основанными на логистической сигмоиде, показали, что последний уровень сети очень быстро насыщается, и выбраться из этой ситуации насыщения глубокой сети очень сложно.

Инициализация весов

К сожалению, простой заменой одной функции активации на другую, проблема обучения глубоких нейронных сетей не решается. Так, последовательные слои с нелинейностью вида tanh поочередно насыщаются: значения нейронов слой за слоем сходятся к 1 или -1, что, как и в случае с обнулением аргумента логистической функции, выводит оптимизацию на градиентное плато и часто приводит к ≪плохому≫ локальному минимуму.

Инициализация Ксавье хорошо работает для симметричных функций, но в современных свёрточных (и не только) архитектурах повсеместно используются и несимметричные функции активации, особенно часто — ReLU. Очевидно, что инициализация весов для этой функции активации должна быть другой.

В 2015 году **Каймин Хе** (Kaiming He) с соавторами опубликовали работу, в которой, помимо нескольких перспективных модификаций ReLU, предложена и подходящая для этой функции активации схема инициализации.

Окончательный вывод о дисперсии отличается от инициализации Ксавье только тем, что теперь нет зависимости от размерности выхода, если приравнять фактор изменения дисперсии единице, то получим:

$$\frac{n_{in}^{(l)}}{2} Var\left(w^{(l)}\right) = 1, Var(w_i) = \frac{2}{n_{in}^{(l)}}., \qquad (225)$$

где l обозначает номер текущего уровня, $n_{in}^{(l)}$ — число нейронов на уровне l.

Инициализация весов

Интересно, что в данной работе для ReLU использовалось не равномерное, а нормальное распределение вокруг нуля с соответствующей дисперсией; такой способ инициализации часто используется на практике в глубоких сетях из ReLU-нейронов:

$$w_i \approx \mathcal{N}\left(0, \sqrt{\frac{2}{n_{in}^{(l)}}}\right). \qquad (226)$$

Авторы данной работы нашли баланс между дисперсиями прямого и обратного шагов: в глубокой сети выходы одного слоя подаются на вход следующему, то есть $n_{out}^{(l)} = n_{in}^{(l+1)}$.

А это значит, что когда возьмем произведение по всем слоям сети, окажется, что все члены посередине сокращаются, и фактор изменения дисперсии становится равным

$$\frac{n_{in}^{(0)}}{n_{out}^{(0)}}\frac{n_{in}^{(1)}}{n_{out}^{(1)}}\frac{n_{in}^{(2)}}{n_{out}^{(2)}}\cdots\frac{n_{in}^{(L-1)}}{n_{out}^{(L-1)}}\frac{n_{in}^{(L)}}{n_{out}^{(L)}} = \frac{n_{in}^{(0)}}{n_{out}^{(0)}}\frac{n_{in}^{(1)}}{n_{out}^{(1)}}\frac{n_{in}^{(2)}}{n_{out}^{(2)}}\cdots\frac{n_{in}^{(L-1)}}{n_{out}^{(L)}}\frac{n_{in}^{(L)}}{n_{out}^{(L)}} = \frac{n_{in}^{(0)}}{n_{out}^{(L)}}$$

где $n_{out}^{(L)}$ — число выходов последнего слоя сети, а $n_{in}^{(0)}$ — число входов первого слоя.

Краткое резюме: для симметричных функций активации с нулевым средним (в основном tanh) следует использовать инициализацию Ксавье, а для ReLU и подобных —инициализацию Хе. Хорошая случайная инициализация позволяет улучшить обучение, к улучшению приводит и нормализация по мини-батчам.

Нормализация по мини-батчам

Идея **нормализации по мини-батчам** (batch normalization), появилась совсем недавно, но оказалось, что она действительно способна улучшить обучение в очень широком спектре архитектур (предложена в статье Сергея Иоффе (Sergey Ioffe) и Кристиана Сегеди (Christian Szegedy) о нормализации по мини-батчам, опубликованной в 2015 году).

При обучении нейронных сетей один шаг градиентного спуска обычно делается не на одном обучающем примере, а на небольшом пакете примеров (мини-батче), который выбирается из всего обучающего множества случайно.

С точки зрения обучения у такого подхода есть сразу несколько преимуществ. **Во-первых,** усреднение градиента по нескольким примерам представляет собой аппроксимацию градиента по всему обучающему множеству, и чем больше примеров в одном мини-батче, тем точнее это приближение. Максимальная точность аппроксимации достигалась бы, если бы размер мини-батча был равен размеру всего обучающего множества, но такая точность обычно вычислительно недостижима.

Во-вторых, глубокие нейронные сети подразумевают большое количество последовательных действий с каждым примером, а современное многопоточное ≪железо≫ позволяет эту последовательность действий выполнять для большого числа примеров в параллельном режиме. При этом, если на очередном шаге градиентного спуска меняются веса одного из первых слоев, то это приводит к изменению распределения весов и активаций на выходах этого слоя.

Нормализация по мини-батчам

А значит, всем последующим слоям надо адаптироваться к по-новому распределенным данным. Эта проблема получила название **внутреннего сдвига переменных** (internal covariance shift).

В классическом машинном обучении данная проблема обычно возникала в том, что распределение данных в тестовой выборке может существенно отличаться от распределения данных в обучающей выборке. Одним из основных методов решения проблемы была нормализация данных в той или иной форме. В классических нейронных сетях нормализация тоже использовалась: известно, что процесс обучения сходится быстрее, когда входы сети ≪отбелены≫ (whitened), то есть их среднее приведено к нулю, а матрица ковариаций — к единичной матрице. В этом состоит и идея нормализации в глубоких сетях: если операцию ≪отбеливания≫ применять ко входам каждого слоя, это позволит избежать проблемы сдвига переменных. Нормализация входов помогает, однако, нормализацию необходимо учитывать и при градиентном спуске.

Для этого вводится слой батч-нормализации. Слой получает на вход очередной мини-батч $B = \{x_1, \dots, x_m\}$, а затем последовательно вычисляет базовые статистики по мини-батчу:

$\mu_B = \frac{1}{m}\sum_{i=1}^{m} x_i$, $\sigma_B^2 = \frac{1}{m}\sum_{i=1}^{m}(x_i - \mu_B)^2$; нормализует входы:

$\hat{x}_i = \frac{x_i - \mu_B}{\sqrt{\sigma_B^2 + \epsilon}}$; вычисляет результат: (227)

$y_i = \gamma x_i + \beta.$

Нормализация по весам

Остался ещё один важный элемент — веса сети. Нормализация весов аналогична нормализации по батчу или по слою, однако коэффициент репараметризации (нормализации) вычисляется с учетом весов очередного слоя, а не его активации:

$$h_t = f\left(\frac{\gamma}{\|w_i\|}\left(w_i^T x + b_i\right)\right) \qquad (228)$$

где w_i — веса линейной комбинации i-го нейрона, b_i — его смещение, а $\|w_i\|$ обозначает евклидову норму вектора весов. После такой репараметризации норма вектора весов оказывается в точности равна γ, и этот параметр нейронная сеть тоже обучает вместе с основными.

Обозначим «новые» веса сети через $v = \frac{\gamma}{\|w\|} w$. Тогда градиент функции ошибки L по γ равен $\nabla_\gamma L = \frac{w^T \nabla_v L}{\|w\|}$,

где $\nabla_v L$ представляет собой обычный градиент по весам сети.

Градиент функции ошибки по исходным весам равен

$$\nabla_w L = \frac{\gamma}{\|w\|} \nabla_v L - \frac{\gamma \nabla_\gamma L}{\|w\|^2} w,$$

Нормализация по весам

и это, подставив выражение для $\nabla_\gamma L$, можно переписать в виде

$$\nabla_w L = \frac{\gamma}{\|w\|} \mathrm{M}_v \nabla_v L,$$

где $\mathrm{M}_v = I - \frac{vv^T}{\|v\|^2}$ — обозначает матрицу проекции на дополнение вектора *v*. По сути, нормализация весов делает две вещи:

- масштабирует градиент с весом $\frac{g}{\|w\|}$|;
- ≪отворачивает≫ градиент от вектора текущих весов.

Оба эти эффекта приближают матрицу ковариаций градиентов к единичной, что позволяет добиться определенных преимуществ при обучении.

В итоге этот механизм позволяет нейронной сети самостоятельно стабилизировать норму градиентов. Кроме того, приятным дополнением оказался эмпирически установленный факт устойчивости к выбору коэффициента обучения.

Самонормализация нейронов

Ещё один подход к нормализации предложен в работе несколько немецких ученых из группы Юргена Шмидхубера. Они разработали так называемые **самонормализующиеся нейронные сети** (selfnormalizing neural networks, SNNs). Оказалось, что можно добавить свойства нормализации непосредственно в функцию активации нейрона, и для этого достаточно просто использовать масштабированную экспоненциальную функцию:

$$\text{SELU}(x) = \lambda \begin{cases} x, & x > 0, \\ \alpha(e^x - \alpha), & x \leq 0. \end{cases} \qquad (229)$$

Все обучение в глубоких нейронных сетях ведется так или иначе при помощи градиентного спуска. Поэтому появление новых идей для изменений, улучшающих сам процесс оптимизации, может привести к очень хорошим результатам. Так получилось с нормализацией по мини-батчам: по сути, ее применение позволило перейти от архитектур глубиной не более двух десятков слоев к сверхглубоким архитектурам, которые сейчас насчитывают сотни и даже тысячи слоев.

Но это был не единственный прорыв в обучении в нейронных сетях. Важнейшую роль для современного глубокого обучения играет и сам процесс градиентного спуска: его современные модификации работают во много раз лучше классических подходов.

Современные персептроны: функции активации

Базовая конструкция персептрона Розенблатта остается актуальной и в настоящее время. Сначала в персептрон поступают входы из данных или предыдущих уровней сети, затем берется их линейная комбинация с некоторыми весами, которые, собственно, и будут обучаться в сети, а потом результат проходит через некоторую нелинейную функцию, без которой, никакой деятельности у нейронной сети не получится. Именно из таких нейронов, состоят все современные нейронные сети. Разница есть только в том, какова, конструкция нелинейности.

Исторически в сигмоидальных нейронах обычно применялась функция активации в виде **логистической сигмоиды:** $\boldsymbol{\sigma(x) = \frac{1}{1-e^{-x}}}$. Эта функция обладает всеми свойствами, необходимыми для нелинейности в нейронной сети: она ограничена, стремится к нулю при $x \to -\infty$ и к единице при $x \to \infty$ и, везде дифференцируема, и производную ее легко подсчитать как $\sigma'(x) = \sigma(x)(1 - \sigma(x))$. Но она такая, не одна.

Гиперболический тангенс:

$$\boldsymbol{tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}}$$

очень похож по свойствам на логистическую сигмоиду.

Современные персептроны: функции активации

Он тоже непрерывен, тоже ограничен (он стремится к —1 при $x \to -\infty$, а не к нулю), и производную от него тоже легко подсчитать через него самого:

$$tanh'(x) = 1 - tanh^2(x)$$

По сравнению с логистической сигмоидой гиперболический тангенс значительно ≪круче≫ растет и убывает, быстрее приближается к своим пределам; например, наклон касательной в нуле у тангенса $tanh'(0) = 1$, а у логистической сигмоиды $\sigma'(0) = \frac{1}{4}$.

Однако есть и более важная разница: для логистической сигмоиды ноль является точкой насыщения, то есть если пытаться обучить значение этой функции в нуле, вход будет стремиться к минус бесконечности, а производная — к нулю, это стабильное состояние. А для $tanh$ ноль — это как раз самая нестабильная промежуточная точка, от нуля легко оттолкнуться и начать менять аргумент в любую сторону. Гиперболический тангенс часто используется в некоторых приложениях нейронных сетей, в частности в компьютерном зрении.

Современные функции активации базируются на так называемых ***rectified linear units*** **(ReLU)**. Функция активации у них кусочно-линейная:

$$\boldsymbol{ReLU(x)} = \begin{cases} \mathbf{0}, \text{если } \boldsymbol{x} < \mathbf{0}, \\ \boldsymbol{x}, \text{если } \boldsymbol{x} \geq \mathbf{0}. \end{cases} \qquad (230)$$

Современные персептроны: функции активации

То же самое можно записать более кратко: $\boldsymbol{ReLU(x) = \max(0, x)}$. Это не совсем современная идея: такие искусственные нейроны использовались еще в начале 1980-х годов в модели многоуровневых сетей Кунихико Фукусимы для распознавания образов, получившей название *Neocognitron*.

ReLU-нейроны эффективнее нейронов, основанных на логистической сигмоиде и гиперболическом тангенсе. Например, чтобы подсчитать производную $\sigma'(x)$ нужно вычислить непростую функцию σ, а затем умножить $\sigma(x)$ на $1 - \sigma(x)$; с тангенсом происходит примерно то же самое. А чтобы вычислить производную ReLU'(x), нужно одно сравнение: если x меньше нуля, выдаем ноль, если больше нуля, — единицу. На практике это означает, что основанные на ReLU-нейронах сети при одном и том же ≪вычислительном бюджете≫ на обучение, на одном и том же ≪железе≫ могут быть значительно больше по числу нейронов, чем сети с более сложными функциями активации. Однако сам по себе этот аргумент мало что значит, надо еще, чтобы новая конструкция действительно чему-то обучалась.

Функция $\boldsymbol{Softplus\ -\ log(1 + e^x)}$, в отличие от сигмоиды, очень похожа на ReLU(x) = max(0, x). Иначе говоря, если посмотреть на ReLU-нейрон теоретически, то видно, что это неплохое приближение к конструкции, которая сильнее и более выразительна, чем обычная логистическая сигмоида. Вероятно, именно этим объясняется успех ReLU-активации в современных нейронных сетях.

Современные персептроны: функции активации

Есть и менее формальное, но тоже любопытное объяснение: оказывается, такая структура активации гораздо точнее отражает то, что происходит с настоящими нейронами в реальном человеческом мозге.

Во-первых, как известно, мозг — это очень энергоемкий орган, он потребляет около 20 % всей энергии, которую тратит человек, при собственной массе около 2 % от массы тела. Поэтому для нашего мозга энергоэффективность, не менее важна, чем собственно вычислительная мощь. Для имеющегося у него огромного числа нейронов мозг крайне энергоэффективен; это достигается, в частности, *разреженностью* активации нейронов: в каждый момент времени активированы от 1 до 4% нейронов в мозге. Но если бы они имели сигмоидную активацию и инициализировались случайно, как во многих нейронных сетях, в каждый момент времени заметно активирована бы была где-то половина нейронов.

Во-вторых, прямые непосредственные исследования функции активации в реальных нейронах и приближенные к биологии модели дают функцию, гораздо больше похожую на ReLU, чем на сигмоиду. В последнее время используются различные модификации и обобщения ReLU, которые сохраняют вычислительную эффективность, но при этом добавляют гибкости в базовую конструкцию.

Современные персептроны: функции активации

Например, ***«протекающий ReLU»*** **(Leaky ReLU, LReLU**: на положительных аргументах функция остаётся той же самой, а на отрицательных становится линейной и при этом отрицательной:

$$\boldsymbol{LReLU(x)} = \begin{cases} \boldsymbol{\alpha x}, & \text{если } \boldsymbol{x < 0} \\ \boldsymbol{x}, & \text{если } \boldsymbol{x \geq 0} \end{cases} \qquad (231)$$

где a — это небольшая положительная константа, например $a = 0{,}1$ (рисунок 21).

Идея здесь заключается в том, чтобы попытаться улучшить обучение, так как если на отрицательной части области определения все градиенты строго равны нулю, эти нейроны не будут обучаться.

Дальнейшим развитием этой идеи стал ***параметризованный ReLU*** **(Parametric ReLU, PReLU).** Эта конструкция выглядит точно так же, как и LReLU, с той лишь разницей, что теперь константу a тоже можно обучать для каждого конкретного датасета.

Современные персептроны: функции активации

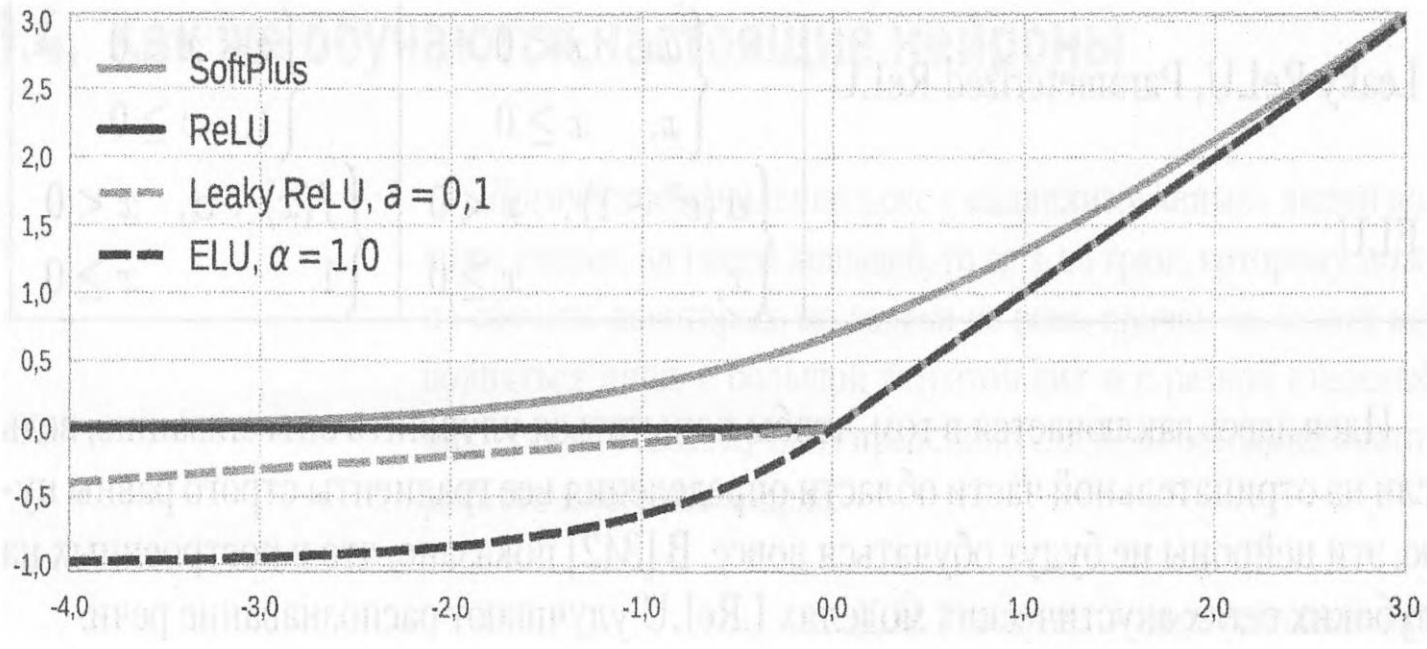
Другой вариант модификации — это ***экспоненциальный линейный нейрон*** **(Exponential Linear Unit, ELU),** в котором на отрицательных значениях аргумента функция активации становится экспоненциальной:

$$\mathbf{ELU} = \begin{cases} \boldsymbol{\alpha}(\boldsymbol{e^x} - \mathbf{1}), & \boldsymbol{x} < \mathbf{0}, \\ \boldsymbol{x}, & \boldsymbol{x} \geq \mathbf{0}. \end{cases} \quad (232)$$

Идея здесь состоит в том, чтобы сочетать в одной функции наличие отрицательных значений и их быстрое насыщение при дальнейшем уменьшении аргумента (это важно, чтобы сохранить разреженность). Есть и другие варианты функций активации (приведены в таблице 3), которые тоже в некоторых областях и экспериментах показывают себя лучше существующих.

Графики функций приведены на рисунке 23.

Современные персептроны: функции активации (рисунок 23)



Современные персептроны: функции активации (таблица 3)

Название функции	Формула $f(x)$	Производная $f'(x)$
Логистическая сигмоида σ	$\frac{1}{1-e^{-x}}$	$f(x) = f(x)(1-f(x))$
Гиперболический тангенс $tanh$	$\frac{e^x - e^{-x}}{e^x + e^{-x}}$	$1 - f^2(x)$
SoftSign	$\frac{x}{1-\lvert x\rvert}$	$\frac{1}{(1-\lvert x\rvert)^2}$
Ступенька	$\begin{cases} 0, & x < 0, \\ 1, & x \geq 0. \end{cases}$	0
SoftPlus	$log(1+e^x)$	$\frac{1}{1-e^{-x}}$
ReLU	$\begin{cases} 0, & x < 0, \\ x, & x \geq 0. \end{cases}$	$\begin{cases} 0, & x < 0, \\ 1, & x \geq 0. \end{cases}$
LeakyReLU, Parameterized ReLU	$\begin{cases} \alpha x, & x < 0, \\ x, & x \geq 0. \end{cases}$	$\begin{cases} \alpha, & x < 0, \\ 1, & x \geq 0. \end{cases}$
ELU	$\begin{cases} \alpha(e^x - 1), & x < 0 \\ x, & x \geq 0. \end{cases}$	$\begin{cases} f(x) + \alpha, x < 0, \\ 1, & x \geq 0. \end{cases}$

Алгоритм градиентного спуска

Коэффициент обучения η — это очень важный параметр в алгоритмах градиентного спуска. Если он будет слишком большой, то алгоритм станет просто прыгать по фактически случайным точкам пространства и никогда не попадет в минимум, потому что все время будет через него перепрыгивать. А если он будет слишком маленький, то во-первых, обучение станет гораздо медленнее, а во-вторых, алгоритм рискует сойтись в первом же локальном минимуме.

Кажется очевидным, что коэффициент обучения должен сначала быть большим, чтобы как можно быстрее прийти в правильную область пространства поиска, а потом стать маленьким, чтобы уже более детально исследовать окрестности точки минимума и в конце концов попасть в нее. Поэтому простейшая стратегия управления коэффициентом обучения так и выглядит: начинаем с большой скорости η_0. и постепенно уменьшаем ее по мере того, как продвигается обучение.

Часто для этого используют или линейное затухание:

$$\eta = \eta_0 \left(1 - \frac{t}{T}\right),$$

или экспоненциальное

$$\eta = \eta_0 e^{\frac{t}{T}},$$

где t — это прошедшее с начала обучения время (число мини-батчей или число эпох обучения), а T — параметр, определяющий, как быстро будет уменьшаться η.

Алгоритм градиентного спуска

Если правильно подобрать параметры η_0 и T, такая стратегия будет работать лучше, чем градиентный спуск с постоянной скоростью. Однако, во-первых, подбор одного параметра η заменён на подбор двух параметров, η_0 и T. Во-вторых, такое постепенное замедление обучения с фиксированными параметрами совершенно никак не отражает собственно форму и характер функции, которую оптимизируем.

Оказывается, можно учесть форму функции, и адаптивные методы градиентного спуска, которые меняют параметры спуска в зависимости от происходящего с функцией, будут работать еще лучше. Идея здесь заключается в том, чтобы вместо глобального коэффициента обучения, который никак не связан с оптимизируемой функцией, попробовать учитывать в коэффициенте обучения непосредственно «ландшафт» функции, данный в изменениях градиентов в различных точках.

Если минимум находится в сильно вытянутом «овраге», шаг градиентного спуска будет направлен от одной близкой и крутой стенки этого оврага к другой. И когда точка попадет на другую стенку, градиент станет направлен в противоположную сторону. Получается, что в процессе обучения алгоритм будет все время прыгать туда-сюда с одной стенки оврага на другую, но к общему минимуму будет продвигаться очень медленно. Такое часто случается поблизости от локальных минимумов, поэтому от подобной проблемы хотелось бы избавиться.

Алгоритм градиентного спуска с моментом

Метод моментов(импульсов) помогает ускорить градиентный спуск в нужном направлении и уменьшает его колебания. Представьте, что материальная точка не просто подчиняется правилам градиентного спуска, а действительно движется по ландшафту оптимизируемой функции. Тогда точка (соответствующая текущему значению вектора весов при обучении), действительно будет стремиться двигаться согласно законам градиентного спуска: ее ускорение будет направлено в сторону, обратную градиенту функции, описывающей поверхность. Но как только точка начнет собственно движение, классический градиентный спуск перестанет быть применим: если точка пришла в очередное положение с какой-то уже ненулевой скоростью, то эта скорость не сможет мгновенно измениться на противоположную, ускорение будет направлено по градиенту, но само движение будет поначалу продолжаться в ту же сторону, в которую и было направлено. Проще говоря, материальные точки обладают инерцией, а их момент движения нельзя изменить мгновенно.

Этот эффект инерции и пытается отобразить метод моментов. Вместо того чтобы двигаться строго в направлении градиента в конкретной точке, алгоритм продолжает движение в том же направлении, в котором двигался раньше; но градиент тоже участвует в формировании окончательной ≪скорости движения≫ вектора аргументов.

Алгоритм градиентного спуска с моментом

Отсюда и название метода: у нашей ≪материальной точки≫, которая спускается по поверхности, появляется импульс, она движется по инерции и стремится этот импульс сохранить. Уравнение для обновления весов, теперь выглядит вот так:

$$w_t = \alpha w_{t-1} + \eta \nabla_\theta E(\theta) \quad (233)$$

$$\theta = \theta - w_t \quad (234)$$

Здесь α — это параметр метода моментов; он всегда меньше единицы, и он определяет, какую часть прошлого градиента мы хотим взять на текущем шаге, а на какую часть будем использовать новый градиент.

Теперь, когда ≪точка≫ катится с горки, она все больше ускоряется в том направлении, в котором были направлены сразу несколько предыдущих градиентов, но будет двигаться достаточно медленно в тех направлениях, где градиент все время меняется.

Метод Нестерова

Метод Нестерова, являющийся модификацией метода моментов, использует при расчете градиента значение функции погрешности не в точке θ, а в точке $\theta - \alpha w_{t-1}$. Это справедливо, если учесть, что αw_{t-1} согласно методу моментов уже точно будет использовано на этом шаге, а значит, и изменившийся градиент разумно считать уже в той точке, куда мы придем после применения момента предыдущего шага.

Математически вектор обновления параметров теперь можно записать так:

$$w_t = \alpha w_{t-1} + \eta \nabla_\theta E(\theta - \alpha w_{t-1}) \quad (235)$$

Для того чтобы иметь возможность адаптировать коэффициент обучения для разных параметров автоматически, были созданы адаптивные методы оптимизации. Несмотря на то, что у них все равно есть свои собственные метапараметры, эти методы, как правило, ведут себя лучше на разреженных данных и более стабильны, чем изменение единого коэффициента обучения в чистом виде.

Адаптивные варианты градиентного спуска

Первый из **адаптивных методов оптимизации, Adagrad**, основан на следующей идее: шаг изменения должен быть меньше у тех параметров, которые в большей степени варьируются в данных, и, соответственно, больше у тех, которые менее изменчивы в разных примерах. Именно из-за этой особенности Adagrad особенно полезен, когда данные сильно разрежены.

Обозначим через $g_{t,i}$ градиент функции погрешности по параметру θ_i

$$g_{t,i} = \nabla_{\theta_i} L(\theta)$$

Тогда обновление параметра θ_i на очередном шаге градиентного спуска можно записать так:

$$\theta_{t+1,i} = \theta_{t,i} - \frac{\eta}{\sqrt{G_{t,ii}+\epsilon}} g_{t,i} \qquad (236)$$

где G_t — диагональная матрица, каждый элемент которой — это сумма квадратов градиентов соответствующего параметра за предыдущие шаги, то есть:

$$G_{t,ii} = G_{t-1,ii} + {g_{t,i}}^2, \qquad (237)$$

а ϵ — это сглаживающий параметр, позволяющий избежать деления на ноль. Интересно, что если не брать квадратный корень в знаменателе, алгоритм действительно будет работать хуже.

Адаптивные варианты градиентного спуска

Поскольку операции ко всем координатам применяются одинаковые, мы можем векторизовать эти выражения так:

$$u_t = -\frac{\eta}{\sqrt{G_{t-1}+\epsilon}} g_{t-1} \quad (238)$$

Здесь и далее мы будем считать, что операции взятия корня и умножения продолжаются на векторы покомпонентно.

Один из плюсов Adagrad состоит в том, что о параметре коэффициента обучения η можно больше не волноваться, так как диагональные элементы G по сути и являются индивидуальными коэффициентами обучения для каждого θ. Поэтому для η обычно используют стандартное значение η = 0,01, но и оно не имеет большого значения, так что настраивать его не обязательно.

А главный минус можно заметить, если обратить внимание на то, что g^2 всегда положительно, а значит, значения G постоянно увеличиваются. Это приводит к тому, что коэффициент обучения порой уменьшается слишком быстро и в итоге становится слишком маленьким, что плохо сказывается на обучении глубоких нейронных сетей. Еще один минус состоит в том, что глобальный коэффициент обучения в Adagrad нужно выбирать руками, самостоятельно, и он может оказаться хорош для одних размерностей, но плох для других.

Адаптивные варианты градиентного спуска

Adadelta — это эффективная модификация алгоритма Adagrad, основная цель которой состоит в том, чтобы попытаться исправить два этих недостатка. Для этого используются две основных идеи. Первая идея довольно проста: чтобы не накапливать сумму квадратов градиентов по всей истории обучения, следует считать их по некоторому окну, а еще лучше по всей истории, но с экспоненциально затухающими весами. Поскольку хранить целую историю предыдущих градиентов очень затратно, реализовать это проще всего уже известным способом — введением параметра инерции.

Теперь на каждый оптимизируемый параметр найдется свой метапараметр, и если обозначить этот метапараметр через ρ, то изменение матрицы G можно будет записать так:

$$G_{t,ii} = \rho G_{t-1,ii} + (1-\rho) {g_{t,i}}^2 \qquad (239)$$

Как и в методе моментов, ρ должно быть, конечно, меньше 1. Все остальное происходит в точности так же, как в Adagrad:

$$u_t = -\frac{\eta}{\sqrt{G_{t-1}+\epsilon}} g_{t-1}. \qquad (240)$$

Теперь G представляет собой экспоненциальное среднее квадратов градиентов, то есть на каждом шаге в G учитывается только часть ≪истории изменений≫, а веса старых значений градиентов быстро уменьшаются.

Адаптивные варианты градиентного спуска

А это именно то, что нужно — уменьшение коэффициента обучения в тот момент, когда изменение целевой функции замедляется, для более тонкой настройки вокруг локального минимума.

Второе изменение, внесенное в Adadelta, предлагает умножить обновление из Adagrad на еще один новый сомножитель: еще одно экспоненциальное среднее, но теперь уже от квадратов обновлений параметров, а не от градиента.

Поскольку настоящее среднее квадратов обновлений неизвестно, то чтобы его узнать, нужно сначала выполнить текущий шаг алгоритма, — оно аппроксимируется предыдущими шагами:

$$\mathbb{E}[\Delta\theta^2]_t = \rho\mathbb{E}[\Delta\theta^2]_{t-1} + (1-\rho)\Delta\theta^2, u_t = -\frac{\sqrt{\mathbb{E}[\Delta\theta^2]_{t-1}+\epsilon}}{\sqrt{G_{t-1}+\epsilon}} g_{t-1}. \qquad (241)$$

Следующий вариант, очень похожий на предыдущие, — это алгоритм **RMSprop;** RMS здесь означает root mean squares, среднеквадратическое отклонение. RMSprop и Adadelta — фактически близнецы-братья, хотя придуманы они были почти одновременно и независимо разными людьми. Оба алгоритма были созданы для решения основной проблемы Adagrad: бесконтрольного накопления квадратов градиентов, которое в конечном итоге приводило к параличу процесса обучения. Такое совпадение неудивительно, ведь оба метода основаны на давно известной классической идее применения инерции, только на этот раз RMSprop использует ее для оптимизации метапараметра коэффициента обучения.

Адаптивные варианты градиентного спуска

Идея RMSProp следующая: вместо полной суммы обновлений G_t будет использоваться усреднённый по истории квадрат градиента. Метод напоминает принцип, используемый в SGD с моментом - метод экспоненциально затухающего бегущего среднего.

Введём обозначение $\mathbb{E}[g_2]_t$ - бегущее среднее квадрата градиента в момент времени *t*. Формула для его вычисления следующая:

$$\mathbb{E}[g_2]_t = \rho\mathbb{E}[g_2]_{t-1} + (1-\rho)g_2. \quad (242)$$

Тогда, подставив $\mathbb{E}[g_2]_t$ в формулу обновления параметров для Adagrad вместо G_t, получим:

$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{\mathbb{E}[g_2]_t + \epsilon}} g_t \quad (243)$$

Основная разница между RMSprop и Adadelta состоит в том, что RMSprop не делает вторую поправку, а просто использует корень из среднего от квадратов (вот он где, RMS) от градиентов:

$$u_t = -\frac{\eta}{\sqrt{G_{t-1} + \epsilon}} g_{t-1} \quad (244)$$

Значение ϵ обычно берут равным 0,9, а η = 0,001.

Адаптивные варианты градиентного спуска

И еще один адаптивный алгоритм оптимизации, который тоже в последнее время часто используется при обучении нейронных сетей — это **Adam.** Как и предыдущие два метода, он является модификацией Adagrad, но использует сглаженные версии среднего и среднеквадратичного градиентов:

$$m_t = \beta_1 m + (1-\beta_1) g_t,\ v_t = \beta_2 v + (1-\beta_2) {g_t}^2,$$

$$u_t = -\frac{\eta}{\sqrt{v_t+\epsilon}} m_t \qquad (245)$$

В исходной статье рекомендуются значения β_1 = 0,9, β_2 = 0,999, $\epsilon = 10^{-8}$.

Adam сходится гораздо быстрее (хоть и с большей дисперсией). Adam — это прямое расширение RMSprop, и, скорее всего, Adam будет в среднем наилучшим выбором. Однако на практике бывали задачи, на которых Adam расходился, а Adadelta давала нормальные результаты.

Можно поэкспериментировать с формулами калибровки алгоритма Adam, здесь просто напрашивается применит метод заглядывания вперёд, как в алгоритме Нестерова.

Адаптивные варианты градиентного спуска

Adamax как раз и есть такой эксперимент. Вместо дисперсии $v_t = \beta_2 v + (1-\beta_2){g_t}^2$ в (245) можно считать инерционный момент распределения градиентов произвольной степени. Это может привести к нестабильности к вычислениям. Однако случай p, стремящейся к бесконечности, работает на удивление хорошо.

$$v_t = {\beta_2}^p v_{t-1} + (1-{\beta_2}^p)|{g_t}^p| \quad (246)$$

Заметьте, что вместо β_2 используется подходящий по размерности ${\beta_2}^p$. Кроме того, обратите внимание, чтобы использовать в формулах Adam значение, полученное в (246), требуется извлечь из него корень: $u_t = {v^{\frac{1}{p}}}_t$. Выведем решающее правило взамен (245), взяв $p \to \infty$, развернув под корнем v_t при помощи (246):

$$u_t = \lim_{p\to\infty} {v^{\frac{1}{p}}}_t = \max\left({\beta^{t-1}}_2|g_1|, {\beta^{t-2}}_2|g_2|, \dots, {\beta^{t-1}}_2|g_{t-1}|, |g_t|\right) \quad (247).$$

Так получилось потому что при $p \to \infty$ в сумме в (247) будет доминировать наибольший член.

Остальные шаги алгоритма такие же как и в Adam.

Адаптивные варианты градиентного спуска

Надам (2015) является аббревиатурой от Нестерова и Адама, оптимизатора. Компонент Nesterov, однако, является более эффективной модификацией, чем его оригинальная реализация.

Во-первых, мы хотели бы показать, что оптимизатор Adam также можно записать так:

$$m_t = \beta_1 m_{t-1} + (1-\beta_1)\frac{\partial \mathbb{E}}{\partial u_t},\ v_t = max\left(\beta_2 v_t, \left|\frac{\partial \mathbb{E}}{\partial u_t}\right|\right) \quad (248)$$

Обновление веса для оптимизатора Адама в алгоритме Надам использует импульс Нестерова, чтобы обновить градиент на один шаг вперед, заменив предыдущее значение m_t в приведенном выше уравнении к текущему значению

$$m_t: \qquad u_{t+1} = u_t - \frac{\eta}{\sqrt{\frac{v_t}{1-\beta_{2,t}}}+\epsilon} * \left(\beta_1 \frac{m_t}{1-\beta_{1,t}} + \frac{1-\beta_1}{1-\beta_{1,t}} * \frac{\partial \mathbb{E}}{\partial u_t}\right) \quad (249)$$

Значения по умолчанию (взяты из Keras): η = 0,002, β_1 = 0,9, β_2 = 0,999, ε= 10^{-7}

Операция свёртки

Многослойный персептрон – это полносвязная сеть, у которой все нейроны одного слоя связаны со всеми нейронами другого, и все нейроны разные. В каждом слое полносвязной сети повторяется одна и та же операция: на вход подается вектор, который умножается на матрицу весов, а к результату добавляется вектор смещений; только после этого к результату применяется нелинейная функция активации.

Такой подход используется независимо от структуры данных, которые предварительно приводятся к векторной форме. Однако многие типы данных имеют свою собственную внутреннюю структуру. Пример такой структуры — изображение, которое обычно представляется как массив векторов чисел. Попробуем модифицировать полносвязную сеть следующим образом.

Что произойдет, если удалить одну связь, то есть сделать нулевым её вес? При разумных ограничениях, когда веса всех связей примерно одинаковы, это не сильно повлияет на результат – нейронов много, каждый вносит свой небольшой вклад, долю такого вклада можно посчитать.

Можно удалять связи с малым вкладом – это процедура ***прореживания*** нейронной сети, при этом происходит экономия вычислений, так как на 0 можно не умножать.

Операция свёртки

Можно ли обучать такую прореженную, или, как говорят, ***неполносвязную*** сеть? Можно, причём теми же алгоритмами градиентного спуска и обратным распространением, при этом необходимо учитывать структуру связей.

Вообще, можно изначально делать сети, в которых связи неполные, а на них накладывать ограничения:

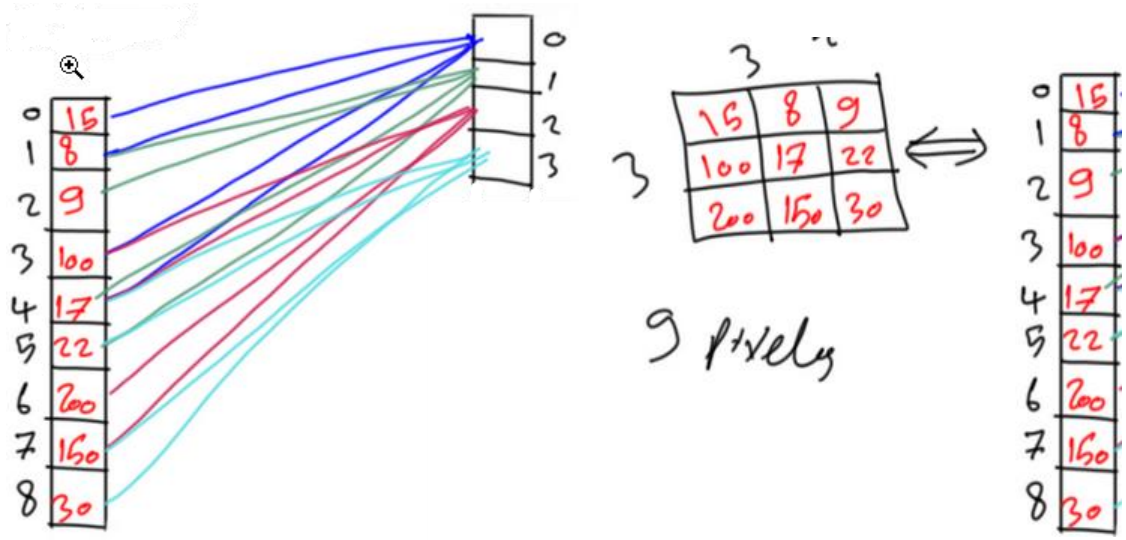
- некоторые связи отсутствуют,
- число связей (входов) нейрона ограничено,
- параметры связи (веса) могут быть одинаковыми у некоторых (или даже всех) нейронов в слое,
- раз многих связей нет, можно ввести правила, по которым нейроны разных слоев соединяются между собой.

Пример: Сделаем одинаковые нейроны, у каждого по 4 входа, подключены как на рисунке 24 а, других связей нет. В персептроне было бы 9 * 4=36 связей и весов, здесь 4 * 4=16 связей, но, так как нейроны мы взяли одинаковыми, то всего 4 разных веса.

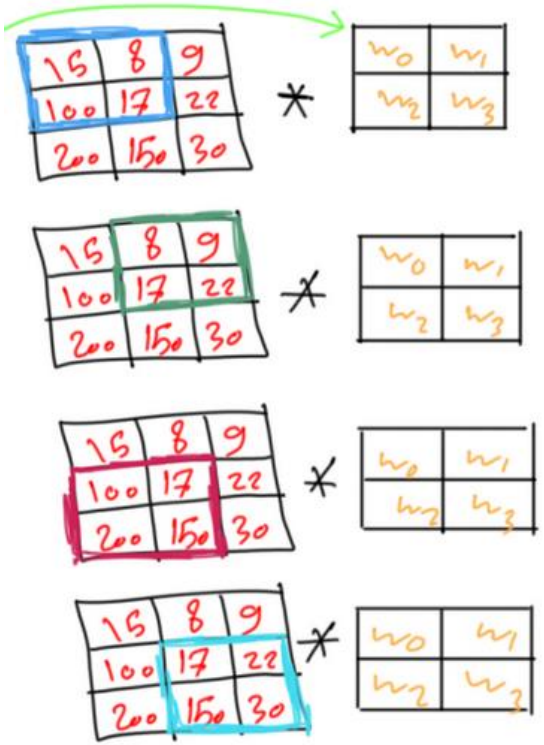
Такой пример можно интерпретировать и по-другому. Переформатируем вход и расположим входы в нейроны не в виде столбца-вектора, а в виде матрицы, это тот же самый слой входов (рисунок 24 б).

Представим их веса (всего 4) как матрицу 2 * 2. Подкрасим те входы, которые используются для конкретного нейрона (нейронов тоже 4, на рисунке 25 сверху вниз).

Пример сети с одинаковыми нейронами по 4 входа (рисунки 24 а и б)



Пример сети с одинаковыми нейронами по 4 входа (рисунок 25)



Двумерная свёртка

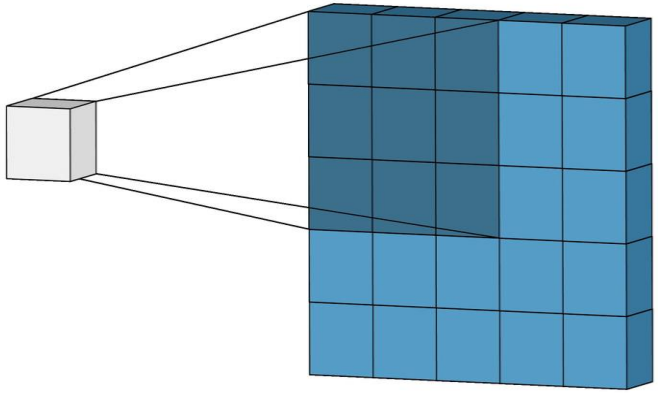
В таком представлении работы нейронов получится, что, некоторое окошко, размером 2 * 2 скользит над входом - матрицей 3 * 3 - принимает входы, над которыми оно висит, умножает их на коэффициенты, складывает произведения и возвращает выход (возможно добавляется смещение и применяется функция активации).

Описанная операция называется (двумерной) **свёрткой** массивов, набор коэффициентов – **ядром свёртки**. Сети, которые основаны на операции свертки – **свёрточные сети**.

В двумерной свертке ядро, т.е. веса - это двумерная матрица, определенной высоты и ширины. Ядро "скользит" над двумерной же матрицей входа (часто это изображение), поэлементно выполняя операцию умножения с той частью входных данных, над которой оно сейчас находится, и затем суммирует все полученные значения в один выходной элемент (пиксель).

Ядро повторяет эту процедуру с каждым положением, над которым оно "скользит", преобразуя двумерную матрицу в другую, все еще двумерную, матрицу выходов (признаков, feature map). Признаки на выходе являются взвешенными суммами (где веса являются значениями самого ядра) признаков на входе. На рисунке 26 показаны примеры свертки

Двумерная свёртка для ядра 3*3 (рисунок 26)



3_0	3_1	2_2	1	0
0_2	0_2	1_0	3	1
3_0	1_1	2_2	2	3
2	0	0	2	2
2	0	0	0	1

12.0	12.0	17.0
10.0	17.0	19.0
9.0	6.0	14.0

Двумерная свёртка

Размер ядра свертки определяет количество признаков, которые будут объединены для получения нового признака на выходе.

В примере, приведенном выше - 5 * 5=25 признаков на входе и 3 * 3=9 признаков на выходе. Для полносвязного слоя (fully connected) размерность весовой матрицы была бы 25 * 9 = 225 параметров, а каждый выходной признак являлся бы взвешенной суммой **всех** признаков на входе. Свёртка позволяет произвести такую операцию со всего 9-ю параметрами, ведь каждый признак на выходе получается не из каждого признака на входе, а только из нескольких, находящегося в "примерно" том же месте.

Свёрточные сети

Так как свёрточные сети изначально использовались для обработки изображений, то основная идея состоит в том, что обработка участка изображения должна происходить независимо от расположения этого участка. Если необходимо узнать на фотографии конкретного человека, то не важно, где он на этой фотографии находится. Узнать человека можно и на сильно обрезанной фотографии, где нет ничего, кроме его лица, это локальная задача, которую можно решать локальными средствами. Взаимное расположение объектов играет важную роль, но сначала их нужно распознать, и это распознавание — локально и независимо от конкретного положения участка с объектом, внутри большого изображения.

Поэтому свёрточная сеть делает это предположение в явном виде: следует покрыть входное изображение небольшими окнами (например, 5x5 пикселов) и выделить признаки в каждом окне небольшой нейронной сетью. Причем признаки надо выделять в каждом окне одни и те же, то есть маленькая нейронная сеть будет одна, входов у нее будет 5 x 5 = 25, а из каждого изображения для нее может получиться очень много разных входов.

Затем результаты этой нейронной сети можно будет представить в виде «изображения», заменяя окна 5 x 5 на их центральные пикселы, и к нему можно будет применить второй свёрточный слой, с уже другой маленькой нейронной сетью, и т. д. В каждом свёрточном слое будет немного свободных параметров, особенно по сравнению с полносвязными аналогами.

Свёрточные сети

Далее необходимо определиться с понятием **канала в изображении**.

Обычно цветные изображения, подающиеся на вход нейронной сети, представлены в виде нескольких **прямоугольных матриц**, каждая из которых задает **уровень одного из цветовых каналов** в каждом пикселе изображения. Изображение размером 200x200 пикселов — это на самом деле 120000 чисел, три матрицы интенсивностей размером 200x200 каждая. Если изображение черно-белое, то такая матрица будет одна. А если это изображение цветное - то матриц обычно 3. Итак, стандартный пример сверточного слоя - принимает трехмерный массив, возвращает трехмерный же, но с числом каналов равным числу фильтров в слое.

Значения каждого признака, которые выделили из окон в исходном изображении, теперь будут представлять собой матрицу. Каждая такая матрица называется **картой признаков** (feature map). В принципе, каналы исходного изображения можно тоже называть картами признаков; аналогично карты признаков очередного слоя можно называть каналами.

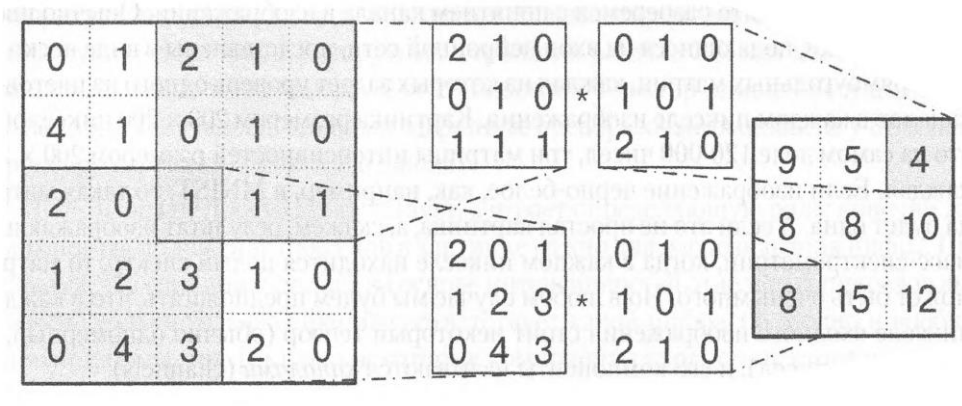
Пример операции свёртки

Иначе говоря, чтобы получить компоненту (i, j) следующего уровня, надо применить линейное преобразование к квадратному окну предыдущего уровня, то есть скалярно умножить пикселы из окна на вектор свёртки. Это проиллюстрировано на рисунке 27, где применяется свёртка с матрицей весов W размера 3x3 к матрице X размера 5x5. Обратите внимание, что умножение подматрицы исходной матрицы X соответствующей окну, и матрицы весов W — это не умножение матриц, а просто скалярное произведение соответствующих векторов. А всего окно умещается в матрице X девять раз, и получается:

$$\begin{pmatrix} 0 & 1 & 2 & 1 & 0 \\ 4 & 1 & 0 & 1 & 0 \\ 2 & 0 & 1 & 1 & 1 \\ 1 & 2 & 3 & 1 & 0 \\ 0 & 4 & 3 & 2 & 0 \end{pmatrix} \times \begin{pmatrix} 0 & 1 & 0 \\ 1 & 0 & 1 \\ 2 & 1 & 0 \end{pmatrix} = \begin{pmatrix} 9 & 5 & 4 \\ 8 & 8 & 10 \\ 8 & 15 & 12 \end{pmatrix}$$

Здесь обозначена свёртка карты признаков X с помощью матрицы весов W через $X*W$, как принято обозначать свёртку в математике.

Пример операции свёртки (рисунок 27)



Если размер свёртки будет выражаться чётным числом, то нет ≪естественного≫ выбора для центра окна, и выбор из четырех вариантов зависит от реализации в конкретной библиотеке. Минимальный размер, при котором окно (фильтр) имеет центр — это размер 3x3. Обычно, для фильтра упоминают только две размерности, отвечающие за ≪рецептивное поле≫ фильтра. На самом деле фильтр задается четырехмерным тензором, последние две размерности которого обозначают число каналов предшествующего и текущего слоя. Так, например, при работе с цветными изображениями на входе есть три канала, передающие соответственно красный, зеленый и синий цвета. Когда говорят, что в первом свёрточном слое ≪фильтр размера 5x5≫, это значит, что в первом слое есть несколько наборов весов (столько, сколько каналов подаются на вход в следующем слое), переводящих тензор размером 5x5x3 в скаляр. Отсюда фильтры размером 1x1 - это просто линейные преобразования из входных каналов в выходные с последующей нелинейностью.

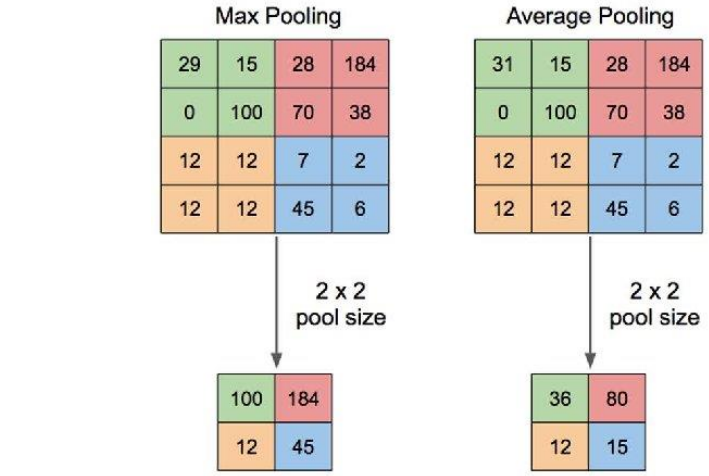
Свёрточные сети

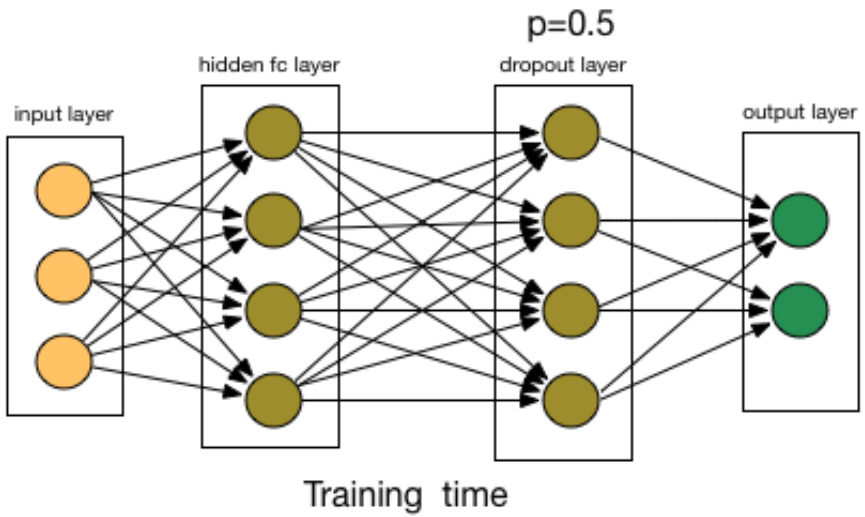
Свёрточная нейронная сеть состоит из множества свёрточных слоев, выходы одних являются входами для других. Свёртка сама по себе - линейная операция, поэтому к результатам свёртки применяют функции активации, обычно одинаковую для всего свёрточного слоя. Почти всегда после свёртки в нейронной сети следует нелинейность, которую можно записать так: $z_{i,j}^{l} = h(y_{i,j}^{l})$.

В свёрточной нейронной сети, кроме слоёв активации (чаще всего ReLU) могут использоваться и другие слои, например:

- полносвязные слои, как в персептронах, часто они являются последними слоями.
- субдискретизирующие слои (пулинг, pooling), которые похожи на свёрточные, но не имеют обучаемых параметров. Пример слоев maxpooling и average pooling показан на рисунке 28, когда из окошка выбирается максимальный элемент, или же находится среднее значение элементов окна, которые и являются выходом. Слой пулинга ≪обобщает≫ выделяемые признаки, потеряв часть информации об их местоположении, но зато сократив размерность. Такие слои помогают существенно уменьшать размер выходных массивов, а значит и число вычислений.
- слои dropout, когда в процессе обучения некоторые связи случайно обрываются. Это заставляет сеть обучаться так, чтобы меньше зависеть от конкретной связи, а больше от их совокупности. Такой процесс сильно помогает в обобщении информации, но он и затрудняет обучение. Может применяться и в персептронных и в сверточных слоях.

Субдискретизация (рисунок 28)





Субдискретизация

Обычно в качестве **операции субдискретизации** к каждой локальной группе нейронов применяется операция взятия **максимума (max-pooling),** нейроны в зрительной коре поступают точно также. Иногда встречаются и другие операции субдискретизации, так первые конструкции группы ЛеКуна использовали для субдискретизации взятие среднего, а не максимума. Однако именно максимум встречается на практике чаще всего и для большинства практических задач дает хорошие результаты, поэтому дальше, под субдискретизацией будет пониматься maxpooling, и формально субдискретизация (в тех же обозначениях, что выше) определяется так:

$$x_{i,j}^{l+1} = \max_{-d \le a \le d, -d \le b \le d} z_{i+a,j+b}^{l}.$$

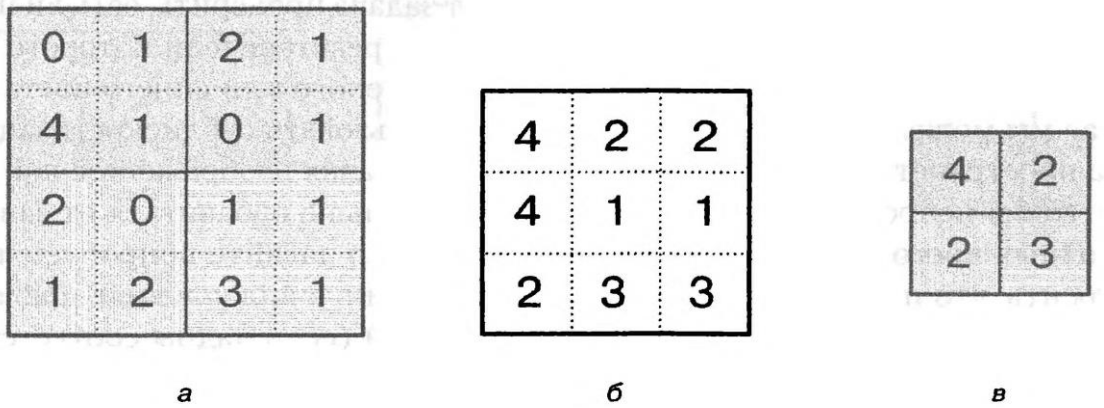
Здесь d —это размер **окна субдискретизации**. Как правило, шаг субдискретизации и размер окна совпадают, то есть получаемая на вход матрица делится на непересекающиеся окна, в каждом из которых выбирается максимум; для d = 2 эта ситуация проиллюстрирована на рисунке 29.

Хотя в результате субдискретизации теряется часть информации, сеть становится более устойчивой к небольшим трансформациям изображения вроде сдвига или поворота.

Итак, стандартный фрагмент свёрточной сети состоит из трех компонентов:

- •свёртка в виде линейного отображения, выделяющая локальные признаки;
- •нелинейная функция, примененная покомпонентно к результатам свёртки;
- •субдискретизация, которая обычно сокращает геометрический размер получающихся тензоров.

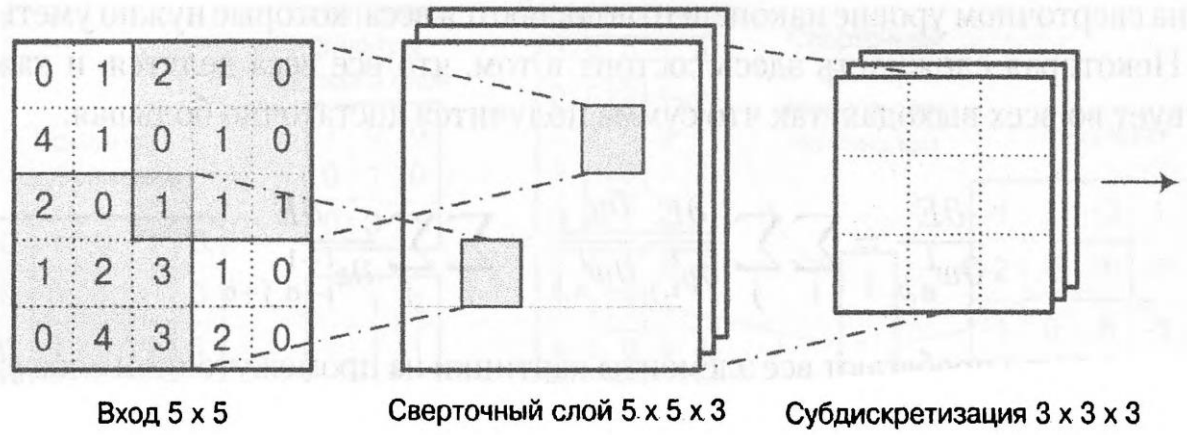
Субдискретизация (рисунок 29)



Пример субдискретизации с окном размера 2x2: а — исходная матрица; б —матрица после субдискретизации с шагом 1; в — матрица после субдискретизации с шагом 2. Штриховка в исходной матрице а — соответствует окнам, по которым берется максимум с шагом 2; в части в — результат показан соответствующей штриховкой.

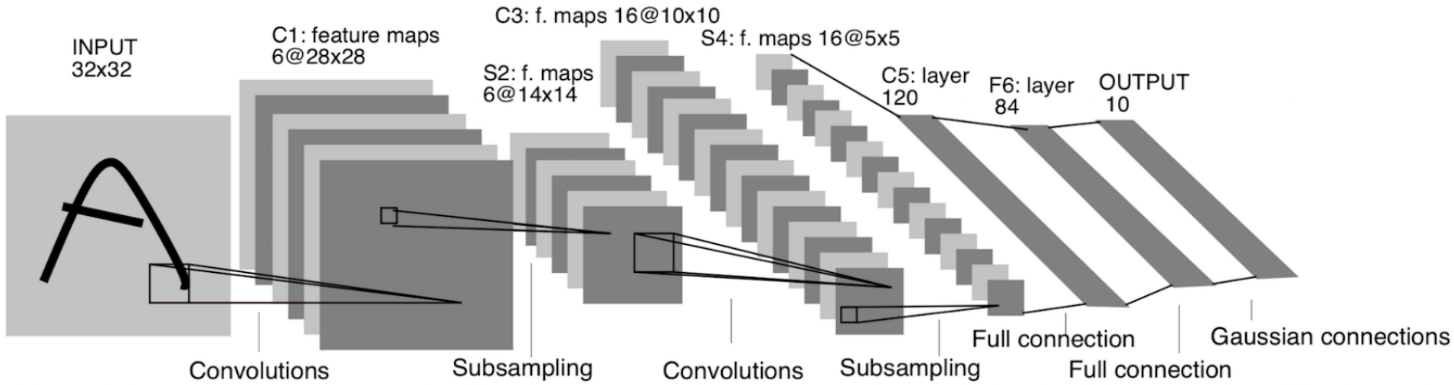
На рисунке 30 изображён стандартный вид фрагмента свёрточной сети.

Схема одного фрагмента сверточной сети (рисунок 30)



Здесь изображён фрагмент сети в виде, в котором он представлен на стандартных изображениях свёрточных сетей в статьях и руководствах. Отдельно выделять нелинейность особого смысла нет, обычно рисуют две части: сначала свёртку, потом субдискретизацию, указывая размерности. По сравнению с изображением на входе размерность тензора увеличилась: свёрточная сеть обычно обучает сразу несколько карт признаков на каждом слое (на рисунке таких карт три). Такая архитектура была применена в знаменитой **сети LeNet-5 Яна ЛеКуна**, которая в конце 1990-х годов показала блестящие результаты на датасете **MNIST** для распознавания рукописных цифр (рисунок 31).

Архитектура сети LeNet-5 (рисунок 31)



После двух фрагментов из свёртки и субдискретизации в сети LeNet-5 следовали три полносвязных слоя с последовательно уменьшающимся числом нейронов, которые совмещали выделенные свёрточными слоями локальные признаки и выдавали собственно ответ. Но надо сказать, что начиная с LeNet и ее более поздних вариантов, набор данных MNIST понемногу потерял свое значение для моделей распознавания образов. В какой-то момент точность превысила **99 %**, затем осталось около **40 ошибок из 10000 примеров в тестовой выборке MNIST**. Сейчас это уже не имеет большого практического значения, и современные модели для анализа изображений сравниваются на других датасетах.

Библиотеки машинного обучения

- *tensorflow* – библиотека для решения задач построения и тренировки нейронной сети,
- *keras* – библиотека для быстрой реализации нейронных сетей, является надстройкой над TensorFlow.
- *pytorch* — фреймворк машинного обучения созданный на базе Torch (Ядро написано на Си, прикладная часть выполняется на LuaJIT, поддерживается распараллеливание вычислений средствами CUDA и OpenMP)

Пример создания простой свёрточной сети

Слои идут последовательно, для чего используется контейнер keras.Sequential()

```
# задаем последовательную модель
model = keras.Sequential( # слои перечисляются ниже
  [
    # слой входов (это не нейроны, а именно входы, т.е. данные), указываем размер shape
    keras.Input(shape=input_shape),
    # первый сверточный слой, указываем число фильтров (32), размер ядра (3 на 3), функцию активации (relu)
    layers.Conv2D(32, kernel_size=(3, 3), activation="relu"),
    # слой пулинга, не обучаемый, но позволит уменьшить размер. указываем размер ядра пулинга (2 на 2)
    layers.MaxPooling2D(pool_size=(2, 2)), #
```

Пример создания простой свёрточной сети

```
 # второй сверточный слой, указываем число фильтров (64), размер ядра (3 на 3), функцию активации (relu)
    layers.Conv2D(64, kernel_size=(3, 3), activation="relu"), ## снова пулинг, после этого слоя размер массива выхода уже не большой
    layers.MaxPooling2D(pool_size=(2, 2)), #
    # слой "выпрямления", когда массив вытягивается в вектор, просто изменяет форму массива,
    # нужен чтобы массивы были подходящей размерности.
    layers.Flatten(), #
    # слой дропаута, будет при обучении случайно удалять связи, указываем вероятность удаления связи от 0 до 1 (0.5)
    layers.Dropout(0.5), #
    # полносвязный слой как в персептроне, размер его равен числу классов, функция активации softmax
    layers.Dense(num_classes, activation="softmax"), #
  ]
)
```

Пример создания простой свёрточной сети

```
model.summary() # информация о созданной сети

Model: "sequential"
_________________________________________________________________
___
Layer (type)                 Output Shape              Param #
=================================================================
===
conv2d (Conv2D)              (None, 26, 26, 32)        320
max_pooling2d (MaxPooling2D        (None, 13, 13, 32)        0
)
conv2d_1 (Conv2D)                  (None, 11, 11, 64)
         18496
max_pooling2d_1 (MaxPooling        (None, 5, 5, 64)  0
2D)
flatten (Flatten)                  (None, 1600)      0
dropout (Dropout)                  (None, 1600)      0
dense (Dense)                (None, 10)                16010
=================================================================
===
Total params: 34,826
Trainable params: 34,826
Non-trainable params: 0
_________________________________________________________________
_
```

Модель свёрточной нейронной сети