ЦИФРОВАЯ КУЛЬТУРА
УНИВЕРСИТЕТ ИТМО

Инструменты обучения нейронных сетей. Оптимизаторы

Высшая Школа Цифровой Культуры
Университет ИТМО
dc@itmo.ru

Содержание

1 Инструменты обучения нейронных сетей

1.1 Введение. Глубина нейронных сетей и некоторые проблемы обучения

Здравствуйте, уважаемые слушатели. В предыдущей лекции мы познакомились с базовой конструкцией слоя в нейронной сети – с так называемым полносвязным слоем. Плодя такого плана слои, эксперементируя с функциями активации между ними, можно получать множество интересных и иногда весьма полезных архитектур. Мы то и дело будем возвращаться к полносвязным слоям, уместно и емко дополняя ими архитектуры, которые встретятся нам с вами далее.

Более того, мы познакомились с основным приемом обучения нейронной сети – с градиентным спуском, и даже с его эвристическим аналогом – стохастическим градиентным спуском. Для их успешного применения мы разобрались с тем, что такое граф вычислений, как осуществлять проход по нему в одном и другом направлениях, и даже на практике «потрогали» матричное дифференцирование, написав нейронную сеть «с нуля».

Понятно, что описанный подход – построение графа вычислений и дифференцирование на нем – практически не зависит от самой архитектуры нейронной сети и, с точки зрения теории, он должен работать всегда, когда только мы умеем «перебрасывать» градиенты через узлы графа. По сути – когда мы можем реализовать процесс backprop'а (процедуру обратного распространения). Но если все так, то тут же назревают вопросы; скажем, почему идеи нейронных сетей так долго «лежали в коробке»? Почему не сделать кучу подряд идущих слоев? Почему не ...?

И правда. Не без некоторого цинизма можно сказать, что сама идея построения графа вычислений не является какой-то очень сложной – способ вычисления производной сложной функции людям известен очень и очень давно (даже в случае матричных аргументов). Сам алгоритм градиентного спуска – тоже не новинка, он впервые был предложен еще до начала 20 века. Ну а идея «навешивания» друг на друга слоев с комбинированием разлчиных функций активации – вовсе не верх изобретательности. Неужели комбинация идей, описанных в трех-четырех предложениях ранее, смогла по-настоящему проявиться лишь в начала 21 века, в 2005-2006 годах, а до этого пылилась на полке или вовсе не осознавалась?

Конечно, это не так. Частично ответ на возникающий вопрос уже был дан ранее: проблема обучения нейронных сетей была сильно связана с отсутствием достаточных вычислительных мощностей компьютеров времен до 21 века, а также, что, наверное, еще существеннее – отсутствием достаточного количества данных, доступных для обучения.

Нельзя не добавить математическую составляющую проблемы. Представьте, что вы обучаете некоторую достаточно «глубокую» (то есть с несколькими скрытыми слоями) нейронную сеть при помощи обратного распространения ошибки. Нейроны последнего слоя, расположенного ближе всего к выходу, достаточно быстро обучатся, а значит, если у них, например, в качестве функции активации выступает классическая сигмоида, их выходы на тренировочных данных будут достаточно близки либо к нулю, либо к единице. Это значит, что значения производной этой функции активации будут что в первом, что во втором случаях близки к нулю и... И мы будем передавать ноль по графу вычислений назад, умножать на этот ноль все остальные градиенты, и все – конец обучению. Это – так называемая проблема затухающих градиентов (vanishing gradients). В итоге, обучившийся слой нейронов просто-напросто «блокирует» собой обучение предыдущих слоев, более ранние слои обучаются либо очень медленно, либо не обучаются вовсе, и чем раньше слой, тем дольше до него доходит информация.

И это еще не все беды. В архитектурах сверточных нейронных сетей, которые просто по определению оказываются «очень глубокими», возникает противоположная проблема – проблема взрыва градиентов (exploding gradients) – проблема экспоненциального увеличения градиентов по мере прохода по слоям сети. Обе описанные проблемы встречаются на практике постоянно, и обе эти проблемы очень долго изучались исследователями в области нейронных сетей.

Сейчас мы умеем бороться с описанными проблемами, и, наверное, львиная доля наших примеров и рассказов будет посвящена принципам этой борьбы – оптимизации или регуляризации в нейронных сетях. Это и регуляризация функции потерь, о которой мы уже вели речь, и модификации метода градиентного спуска, о которых мы поговорим в этой лекции, и так называемый дропаут – случайное отключение нейронов, нормализация мини-батчей, правильная начальная инициализация весов, и так далее. Но... Обо все по порядку.

Говоря про планы, нам кажется, что мы либо не ответили, либо лишь косвенно ответили на следующего рода важный вопрос: зачем? Зачем нужны глубокие нейронные сети? Почему не остановиться на одном слое? Скажем, классическая в математике теорема, одновременно и независимо полученная Г. Цыбенко, К. Фуханаши и К. Хорником утверждает, что любую непрерывную функцию многих переменных (а это очень много) можно сколь угодно точно приблизить нейронной сетью с одним скрытым слоем. Зачем, казалось бы, усложнять?

Несколько голословно мы частично дадим ответ в следующей форме: глубокие хорошо настроенные сети часто позволяют приблизить ту же самую зависимость более эффективно, нежели их неглубокие аналоги. Не вдаваясь в

некоторые боле хитрые детали, отправляем заинтересованных слушателей к интересной работе, в которой наглядно показано как слой с ReLU активацией в каком-то смысле «сворачивает» части пространства, отождествляя их между собой так, что при «разворачивании» пространства обратно разделяющие поверхности превращаются в весьма сложные конструкции в пространстве исходных входных векторов.

Второй момент, который отличает глубокие нейронные сети от неглубоких заключается в так называемой распределенности. Дело в том, что каждый слой глубокой нейронной сети состоит не из одного нейрона, а из нескольких, и их различные комбинации создают настоящий экспоненциальный взрыв в пространстве входов. Мы уже иллюстрировали работу однослойной сети в случае задачи классификации – она разбивает пространство признаков гиперплоскостями на множество частей. Представьте, что мы имеем дело с плоскостью и тремя прямыми, они способны разбить пространство аж на 8 (минимально – на два) различных регионов, а что если добавить еще один слой? Тогда картинки получатся уж слишком сложными.

В общем, надеемся, мы убедили вас в том, что (умеренная) глубина нейронной сети – полезное качество. Что же, теперь настало время перейти непосредственно к инструментам обучения нейронных сетей, а точнее – к инструментам автоматического дифференцирования, ведь именно на дифференцировании, как мы знаем, и строится обучение.

1.2 Autograd

Давайте разберемся, какие в настоящее время существуют средства для облегчения процесса разработки моделей, как они работают и как ими пользоваться.

Как мы уже не раз упоминали, с точки зрения математики, нейронная сеть – это комбинацией различных слоев, каждый из которых представлен матрицей весов и правилами его применения. При таком подходе функцию активации можно тоже рассматривать как слой, просто с другими весами (ведь некоторые функции активации, такие, как ParametrizedReLU, имеют обучаемые параметры), другими правилами применения этого слоя и, как следствие, особым способом подсчета градиента. Данные слои могут комбинироваться на различный манер, образуя структуру сети. Структура сети, на самом деле – синоним графа вычислений. Пример вы можете видеть на рисунке 1

Так как многие слои – строительные блоки нейронных сетей – стандартизированы и реализуются одинаковым образом, были созданы специальные библиотеки, которые за внешним интерфейсом, удобным для программиста, скрывают монотонный (и весьма оптимальный) код описания прямого прохода и обратного распространения ошибки через данный граф. Названия самых

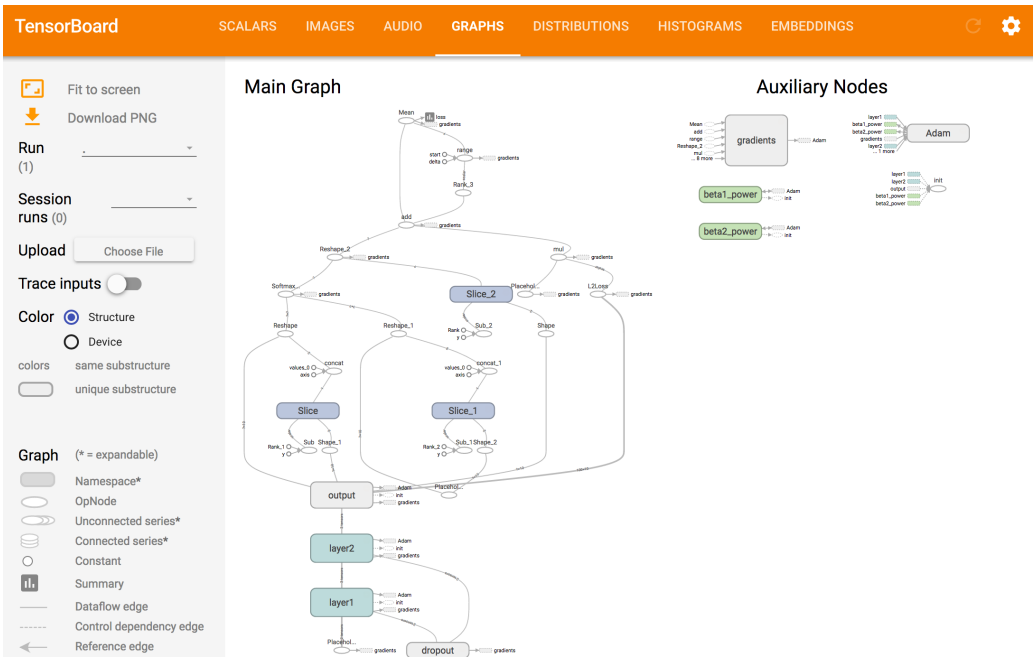

Рис. 1: Схема нейронной сети

популярных библиотек вы можете увидеть на рисунке 2.

Мы с вами обратимся к рассмотрению TensorFlow от команды Google Brain. Эта библиотека дает возможность построить необходимый вам граф вычислений из различных слоев, добавлять на эти слои ограничения, изменять параметры и функции активации, использовать различные оптимизаторы, а также производить обучение моделей на графических или иных специализированных ускорителях.

Кроме того, так как обучение – это проход туда-сюда по графу вычислений, то одной из важнейших причин использования таких библиотек является наличие так называемого autograd'a – движка автоматизированных вычислений производных, который следит за тем, как слои и компоненты соединяются друг с другом и реализует прямое и обратное распространение ошибки. Это обеспечивается тем, что для каждой операции на графе вычислений, построенном с помощью данных библиотек, хранится информация о том, как вычислять градиент, позволяя в дальнейшем autograd'у произвести необходимые операции. Таким образом, инженер избавлен от необходимости самостоятельно писать однотипные функции вычисления градиента и может сосредоточиться на более высокоуровневых вещах, таких как архитектура сети или используемые гиперпараметры.

В рамках данного курса мы будем использовать библиотеку TensorFlow, о которой сейчас и поговорим.

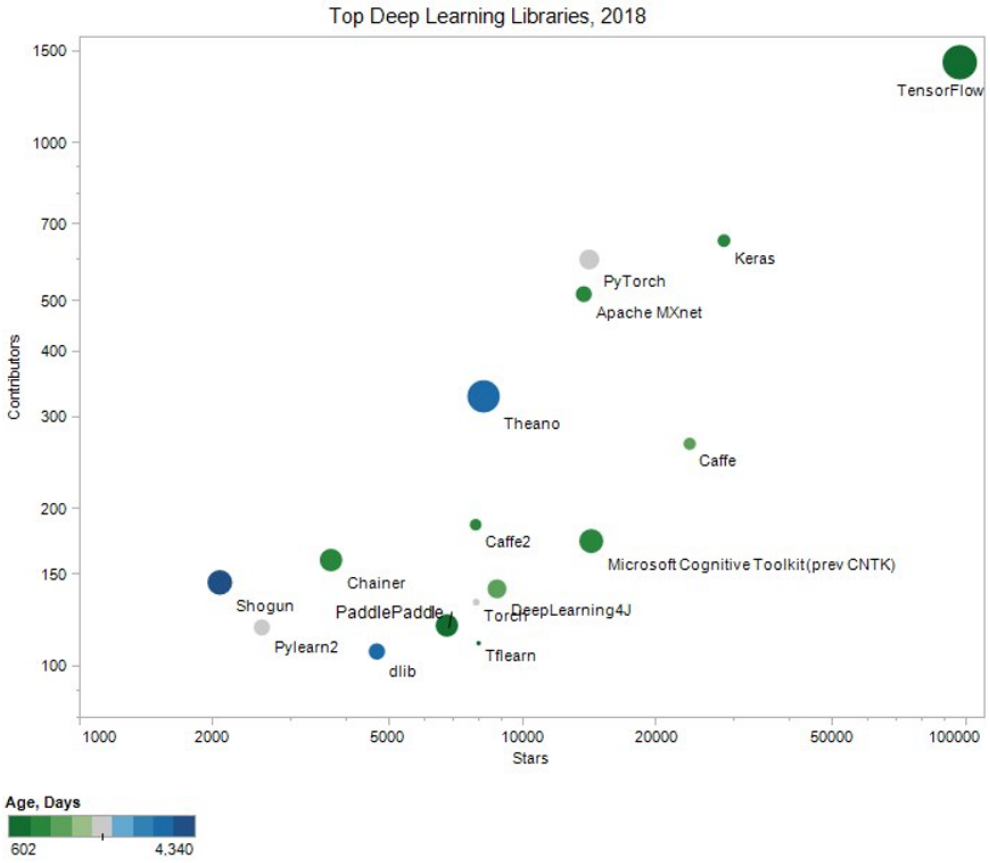

Рис. 2: Популярные библиотеки для обучения нейронной сети

1.3 Tensorflow

TensorFlow – это открытая программная библиотека для машинного обучения, которая была разработана для построения и тренировки искусственных нейронных сетей. Данная библиотека была разработана компанией Google, в дальнейшем выложена в открытый доступ, и в настоящее время является одной из самых популярных библиотек для построения и обучения нейронных сетей. В этом фрагменте мы с вами рассмотрим основные идеи и понятия, используемые в анонсированном продукте, и разберемся с тем, как эта библиотека облегчает построение и тренировку сетей.

В первую очередь, TensorFlow – это пакет, предназначенный для упрощения разработки сетей, проверки гипотез, тренировки моделей, а также для исследовательских целей. Как уже отмечалось и что, наверное, достаточно очевидно – это то, что «архитектура» (в смысле создания построек) нейронных сетей сочетает в себе множество «строительных блоков» – различного вида слоев, параметров этих слоев, оптимизаторов и многих других элементов, которые можно по-разному комбинировать друг с другом, видоизменять, настраивать и дополнять. Данные блоки являются более-менее стандартными и нет никакого смысла каждый раз описывать их в коде с нуля. TensorFlow

как раз-таки и представляет собой библиотеку такого рода блоков вкупе со средствами их обучения. Использование этой библиотеки не только экономит время на разработку моделей, но и снижает вероятность ошибок при написании кода. Более того, надо признать, библиотека и операции в ней написаны очень и очень оптимально, так что не каждый разработчик способен добиться кастомных вычислительных скоростей самостоятельно, об этом чуть позже.

Вторая важная деталь, присущая многим фреймворкам глубокого обучения, и TensorFlow в том числе – наличие уже ранее анонсированного autograd'a. При создании нейронной сети средствами TensorFlow все (самые разнообразные) блоки вашей сети могут быть автоматически связаны друг с другом, для них будут вычислены градиенты, и будет определено направление их «течения».

В третьих, повторяясь, TensorFlow – это очень эффективно написанная библиотека. Данный фреймворк написан с учетом возможности выполнения расчетов на разных процессорах, с разными доступными наборами инструкций, а также с учетом возможности выполнения на графических ускорителях или даже специализированных тензорных ускорителях. Кроме того, внутри этой библиотеки скрыто огромное количество специализированного кода для того, чтобы максимально эффективно обрабатывать данные. Фреймворк предоставляет API для работы с ним из разных языков программирования, таких как C++, Python, Java. Кроме того, существует написанная сообществом поддержка многих других языков.

Таким образом, любой проект может быть виртуально разделен на две части – «внешняя» – язык программирования, на котором вы задаете описание вашей модели и который взаимодействует с внешним API фреймворка, и «внутренняя» – часть фреймворка, которая будет непосредственно создавать модель и транслировать ваши команды в необходимый код для ее обучения и использования. Такая гибкость позволяет вам работать (благодаря коммьюнити data scientist-ов) с использованием практически любого языка программирования, того, который вам нравится, и при этом почти не терять в скорости тренировки и последующего использования модели.

Теперь обратимся к **внутреннему устройству фреймворка**. Внимательный слушатель, услышав название TensorFlow, может сразу задаться резонным вопросом: что это за поток тензоров?

Что же, в первую очередь стоит разобраться с тем, что такое «тензор». В принятой в этой сфере терминологии (отличной, от терминологии математики), тензор – это многомерный массив данных, доступный для обработки и оперирования. Данный массив определяется одним из доступных типов данных (например, float32), и размерностью. В рамках фреймворка все тензоры иммутабельны – в случае изменения данных создается новый тензор. Тензоры – не единственный объект, который можно использовать. Есть еще и

так называемые «зубчатые массивы» (они же – jagged array) – массивы, у которых не все элементы в рамках одной размерности обладают одинаковой длиной, и разреженные массивы.

Переменная в рамках фреймворка TensorFlow – это объект для хранения тензора, значения которого изменяются (конечно, в процессе обучения). Обычно переменные используются для хранения, например, параметров модели – весов, которые как раз-таки и изменяются в процессе обучения.

Важно обратить внимание на часто используемое понятие – «eager execution», оно же – «моментальное исполнение». Происходит оно вот откуда. Оказывается, что взаимодействие с TensorFlow может быть организовано в двух режимах. Первый режим – это режим моментального исполнения. В этом режиме все ваши команды и взаимодействия, направленные на тензоры, будут тут же исполнены, и тут же будет посчитан результат. Второй режим – это режим задания графа вычислений. В этом режиме вы определяете последовательность вычислений над тензорами и переменными, а затем, уже отдельной инструкцией, заполняете необходимые тензоры и запускаете процесс вычисления графа.

Чаще всего в процессе разработки нейронных сетей используется второй режим работы. Первый же тоже не нужно недооценивать. Он позволяет вам исследовать структуру сети, выполнять код и анализировать вывод модели на наличие ошибок и несостыковок. Иными словами, в первом режиме очень удобно производить отладку написанного вами кода. В дальнейшем, обсуждая TensorFlow и фреймворк Keras, чаще всего мы будем работать во втором режиме – сначала определение графа, а потом уже запуск вычислений.

Фреймворк TensorFlow изначально создавался для общих целей, то есть как для обучения и тренировки простых моделей, так и для исследовательских задач и построения сложных архитектур. Вследствие этого, его структура модулей и функций построена в достаточно общем стиле и для использования в рамках простых задач приходится писать множество повторяющихся вещей, таких как функция тренировки, предоставление коллбэков, и так далее. Для упрощения работы мы с вами будем использовать часть TensorFlow под названием Keras – специальный модуль, создающий абстракции более высокого уровня, но дающий меньшую гибкость при построении и использовании модели. Чаще всего, если вам необходимо выполнить инженерную задачу, возможностей модуля Keras вам будет достаточно. Однако, при выполнении исследовательских задач или построении очень сложных архитектур, вам, возможно, придется обратиться к изначальным компонентам фреймворка TensorFlow.

Чтобы не быть голословными, давайте обратимся к некоторым примерам написания сетей на TensorFlow.

1.4 Keras

Итак, как мы уже сказали, Keras – это достаточно удобная надстройка (инженерного плана) над TensorFlow. При помощи этой надстройки можно создавать даже достаточно хитрые архитектуры, по сути, одним и тем же способом – навешивая некоторые «строительные блоки» друг на друга.

Понятно, что такое удобство – удобство использования заранее написанных, оптимизированных черных ящиков – может оборачиваться проблемой в случае, когда вы хотите сделать что-то совершенно нестандартное и даже, быть может, исследовательское. Оставляя исследовательские детали «на потом», обратимся к практическому применению Keras, попутно описав всевозможные возможности этой надстройки.

1.5 Гиперпараметры и их подбор

Несмотря на кажущуюся автоматизацию области МО и ИИ, очень многие решения в процессе работы над той или иной задачей все равно принимает data scientist. Какие-то решения автоматизировать достаточно сложно: например, выбор правильной метрики, относительно которой в дальнейшем считать обучение успешным или нет. Ведь часто здесь необходим опыт и комментарии, относящиеся к предметной области, а не просто сухие числа. Да и гонка-то не за числами, выдаваемыми разными метриками.

В то же время, есть и такие решения, которые можно принимать автоматизированно. Многие алгоритмы машинного обучения и, в частности, нейронные сети, оперируют понятием «гиперпараметры». Гиперпараметры – это параметры модели, которые инженеру необходимо задать самостоятельно перед началом обучения, которые не учатся (не изменяются) в процессе обучения основной модели, но которые в ней (в модели) непосредственным образом участвуют, от которых она (модель) зависит. Мы знаем примеры. Коэффициент регуляризации при использовании l_1 или l_2 регуляризации во время составлении функции потерь для решения задачи классификации – вполне себе гиперпараметр. Приведем и еще классический пример, касающийся решающих деревьев: максимальная глубина дерева, минимальное количество экзмепляров данных в листе – все это тоже гиперпараметры.

В приложении к нейронным сетям в качестве гиперпараметров часто выступают следующие, но не единственные: архитектура сети, то есть выбранные слои и тип их связи; гиперпараметры каждого слоя отдельно, такие как количество нейронов, функции активации, и многое другое; параметры оптимизатора — например, коэффициент скорости обучения; параметры обучения и масса всего еще. Конечно, с этим надо что-то делать, но что?

Итак, перед началом обучения модели все описанные (и неописанные) гиперпараметры инженер должен задать самостоятельно. Некоторые из этих

гиперпараметров, как уже отмечалось, выбираются инженером из его опыта или общепринятых практик – например, архитектура сети. Однако, такие гиперпараметры как количество нейронов в слое или функция активации слоя, сложно угадать – в разных задачах наиболее оптимальным выбором будут разные варианты, и выбор этих вариантов вовсе неочевиден. Поэтому ничего не остается, как подобрать эти параметры. Как? Конечно, автоматически: натренировать несколько моделей с разными комбинациями параметров и выбрать из них лучшую.

Существует несколько подходов к этому автоматическому подбору. Наверное, самый известный подход – это GridSearch (поиск на сетке). В рамках этого подхода для каждого параметра указывается весь список значений, который нужно проверить. Тем самым, если у нас имеется n настраиваемых параметров, каждый из которых может принимать $k_1, k_2, ..., k_n$ значений, соответсвенно, то нам нужно вычислить качество модели аж $k_1 \cdot k_2 \cdot ... \cdot k_n$ раз (по одному разу для каждого набора возможных параметров). Понятно, что в этом и кроется главный недостаток такого, казалось бы, очень наглядного и очевидного подхода – экспоненциальный рост количества вычислений в зависимости от количества настраиваемых параметров. Вы же понимаете, что при каждом наборе параметров модель нужно не просто вычислить один раз, но и обучить, а процесс обучения может быть и сам очень и очень долгим.

Альтернативным подходом к простому перебору является очень популярная в наше время байесовская оптимизация. Не претендуя на строгость изложения, постараемся на пальцах дать вам представление об основной идее подхода. Пусть $G(x_1, x_2, ..., x_n)$ – это функция от гиперпараметров нашей модели, на выходе выдающая качество обученной с этими гиперпараметрами модели (по сути – выдающую какую-то метрику, которую мы считаем удачной). Байесовский подход опирается на предположение, что функция G является непрерывной, а значит малые изменения аргументов приведут к малым изменениям значения функции. Учитывая это предположение, алгоритмы байесовской оптимизации стараются найти области наиболее удачных значений интересующих нас метрик, а потом найти оптимальную точку уже в найденной, суженной области.

В рамках данной лекции мы с вами возьмем один из самых популярных пакетов для байесовской оптимизации — Optuna — и покажем некоторые приемы его применения.

1.6 Under- and Over-fitting

Теперь поговорим про еще несколько терминов, постоянно встречающихся как в теории нейронных сетей, так и в машинном обучении в целом. Термины эти – недо- и пере-обучение (under- и over-fitting). Идейно представление об озвученных терминах можно понять, посмотрев на рисунок 3.

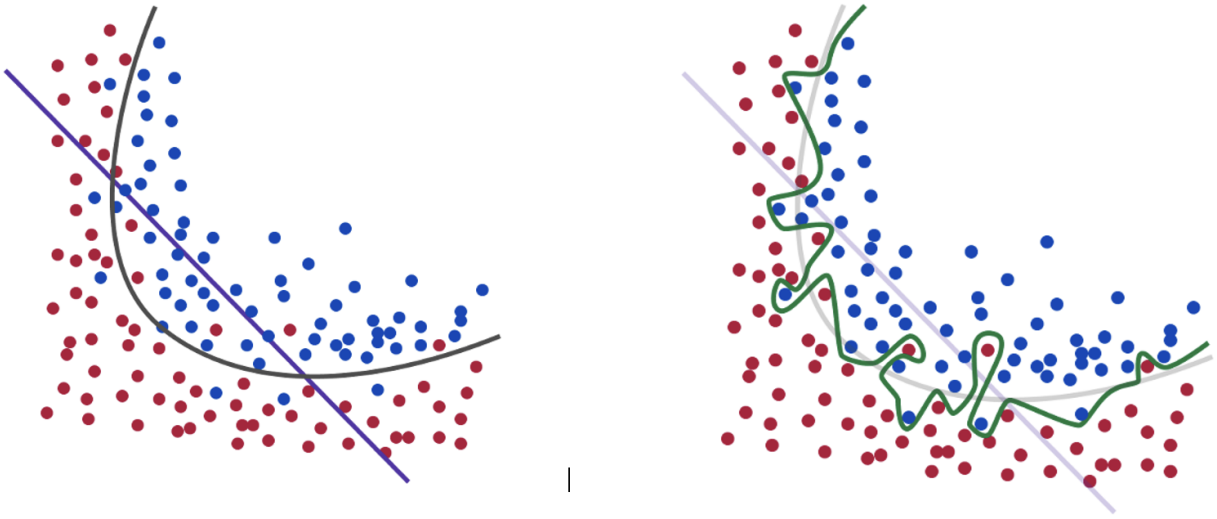

Рис. 3: недо и пере-обучение

Начнем с недообучения (фиолетовая прямая на рисунке). Итак, недообучение – это ситуация, когда модель не смогла получить достаточного количества информации из тренировочного набора данных: модель плохо описывает как тренировочный, так и валидационный наборы. Наверное, можно провести аналогию со студентом, который перед экзаменом успел выучить только часть заявленных тем. Вероятно, он не сможет ответить ни на вопросы многих билетов (тренировочный набор), ни на какие-то дополнительные вопросы по сути материала (валидационный набор).

Увидеть, что мы находимся в режиме недообучения, достаточно просто, рисунок 4. Обычно этому свидетельствуют близкие значения метрик на тренировочном и тестовом наборах данных, медленно и совместно уменьшающиеся (или увеличивающиеся) к нравящимся нам значениям. Причин такого поведения может быть несколько:

1. может быть, модель учится недостаточное время – можно попробовать увеличить количество эпох обучения

2. может быть, модель недостаточно сложная (например, мы пытаемся приблизить границу параболического типа прямой, рисунок 3) – нужно внимательнее разобраться в сути данных и дать модели большую свободу (гибкость) при обучении

3. может быть, используемый подход в целом неверен – нужно менять подход.

Очень часто многое зависит от выбора скорости обучения (learning rate), рисунок 5. Этот параметр, условно, отвечает за то, какой длины шаг вы совершаете в процессе оптимизации. Понятно, что этот параметр важен, ведь

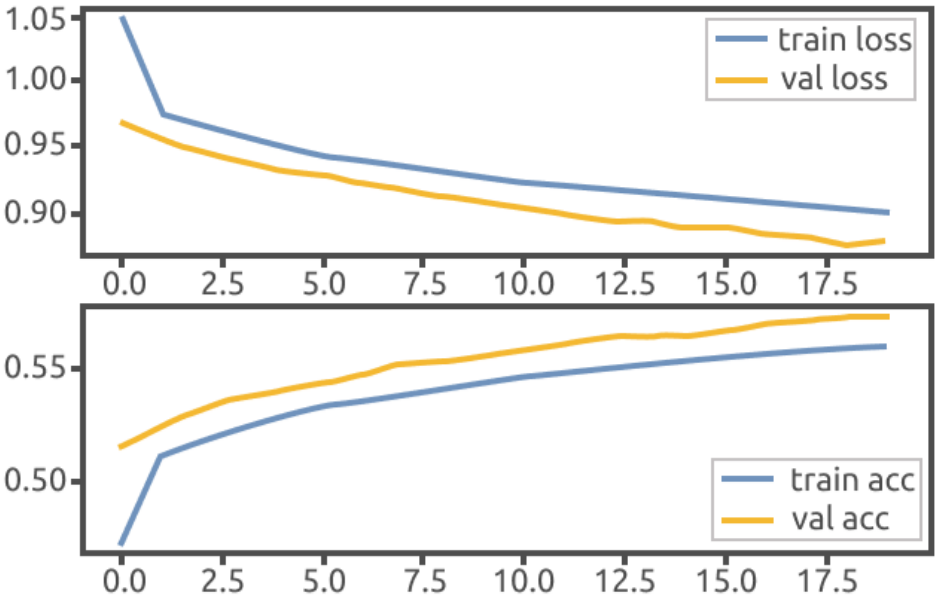

Рис. 4: недобучение

если делать шаг маленьким, а начальная инициализация не очень удачна, то можно либо попасть в локальный минимум, который сильно отличается от глобального, либо до глобального просто не добраться – времени не хватит (синяя кривая). С другой стороны, большая величина шага может повлечь то, что минимум будет просто пропущен, а сам процесс – разойдется (желтая кривая).

Рекомендации здесь можно дать следующие: логично в начале обучения брать learning rate побольше и уменьшать его с ростом числа эпох. Полезно также изменять его в случае попадания на плато: ситуация, когда функция потерь все еще уменьшается на тренировочном наборе данных, но постоянна на валидационном (зеленая кривая). Мы еще будем говорить про так называемый learning rate schedule в последующих лекциях.

Теперь поговорим про более плохую ситуация – про переобучение (зеленая кривая). Переобучение – это, по своей сути, слишком тщательное заучивание тренировочного набора данных. При переобучении модель может выдавать очень хорошие метрики качества на тренировочном наборе данных, но обладать весьма плохой генерализующей способностью, то есть выдавать плохие метрики на тестовом или валидационном наборе (зеленая кривая на рисунке), рисунок 6. Наверное, можно провести аналогию со студентом, который сдает экзамен, наизусть выучив конспект, но так и не разобравшись в материале – весьма бесполезное дело.

Как же увидеть, что мы попали в режим переобучения? Для этого полезно контролировать изменение значений функции потерь как на трениро-

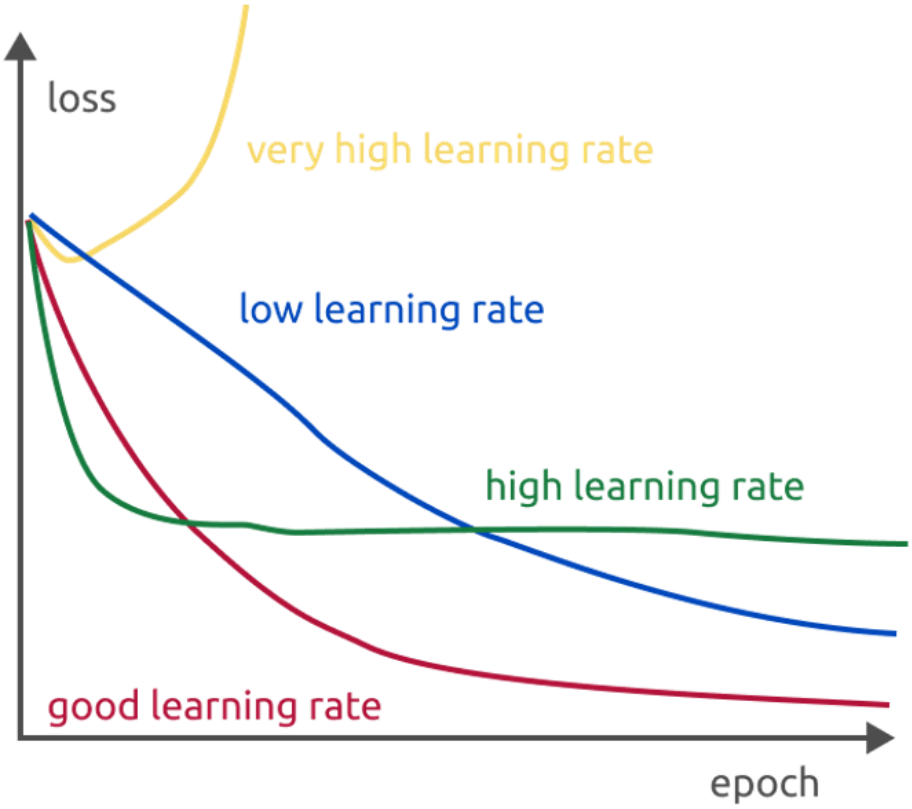

Рис. 5: learning rate

вочном наборе данных, так и на валидационном, а также значения метрик качества на этих множествах. Обычно, находясь в режиме переобучения, значения продолжают улучшаться на тренировочном наборе данных, но перестают меняться или даже начинают ухудшаться на валидационном, рисунок **??**

Борьбу с переобучением вести куда сложнее – обычно используются разного рода регуляризации, частично уже упомянутые нами, а частично – изучаемые в следующих разделах курса.

Давайте теперь посмотрим на введенные понятия на практике.

2 Оптимизаторы в НС

2.1 Введение

В этой части лекции мы остановимся подробнее на тех популярных алгоритмах оптимизации, которые на данный момент повсеместно используются в нейронных сетях. Наверное, нет нужды мотивировать важность выбора корректного алгоритма оптимизации, ведь, по сути, само обучение НС – это оптимизация. Каждый раз, как только очередной батч данных проходит че-

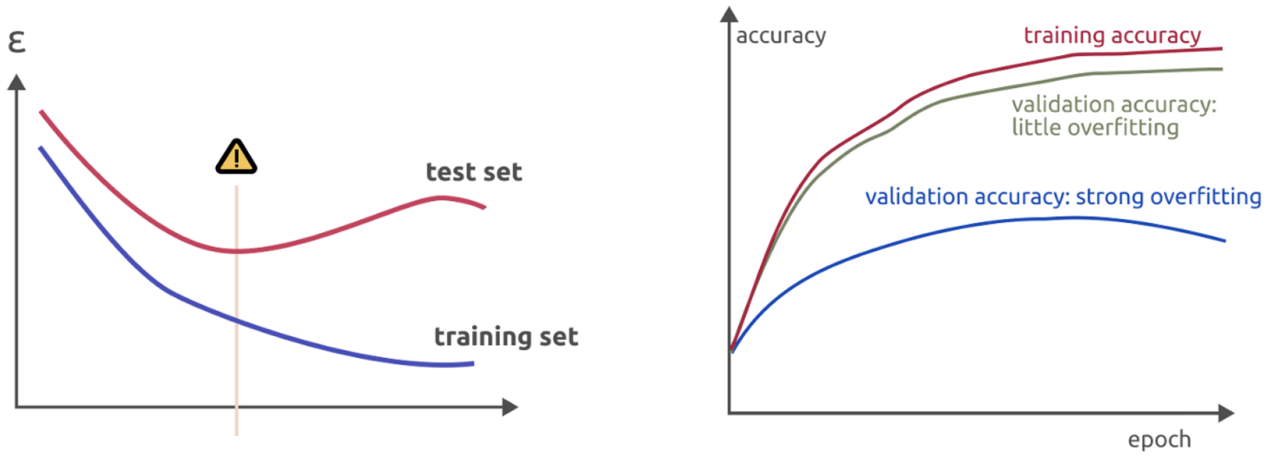

Рис. 6: overfitting

рез нейронную сеть, а та, в свою очередь, выдает предсказания для этого батча, приходится отвечать на следующий вопрос: как использовать информацию о различиях между тем, что выдала нейронная сеть, и известными, истинными значениями предсказанных переменных, для последующего изменения весов модели с целью построения более качественных предсказаний в будущем? Этим вопросом как раз-таки и занимается алгоритм оптимизации, минимизирующий рассматриваемую функцию потерь. Ну что, давайте поговорим про некоторые популярные алгоритмы, а также сравним их между собой.

2.2 SGD и скорость обучения

SGD (stochastic gradient descent, keras.optimizers.SGD) – стохастический градиентный спуск – классический алгоритм оптимизации. В принципе, о нем мы уже говорили ранее. Он аналогичен градиентному спуску, но на каждом шаге использует не весь набор тренировочных данных для составления функции потерь, и, соответственно, вычисления градиента, а лишь некоторый батч (часть данных). Важным параметром является как размер батча, так и шаг, или, так называемый, learning rate.

Скорость обучения (тот самый learning rate) – это чрезвычайно важный параметр. Если скорость будет слишком большой, то, скорее всего, алгоритм будет «прыгать» по случайным точкам пространства и, тем самым, достигнет минимум с весьма небольшой вероятностью. Если же скорость обучения будет маленькой, то описанная проблема не возникает, но возникает другая, не менее интересная проблема: алгоритм останавливается чуть ли не в первом попавшемся локальном минимуме, значение в котором может быть сильно больше, чем значение в «более хорошем» локальном минимуме, не говоря уже о глобальном, рисунок 7.

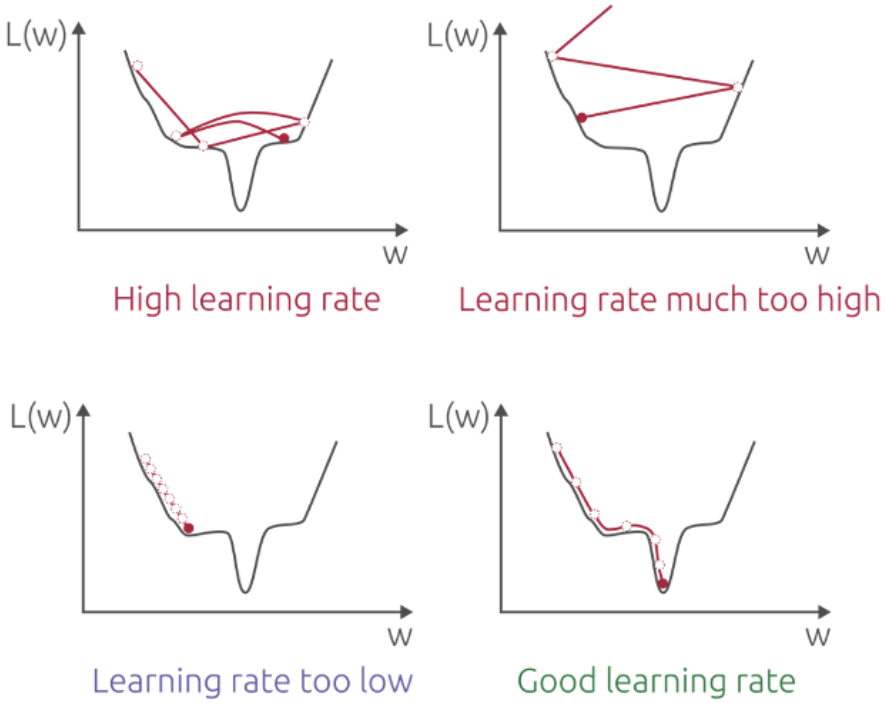

Рис. 7: Выбор шага

Кажется интуитивно понятным, что скорость обучения сначала должна быть достаточно большой, чтобы быстро прийти в нужную область пространства, а затем стать сильно меньше, чтобы детально исследовать окрестность точки минимума и, в конце концов, не пропустить его (и попасть в него). Поэтому простейшая стратегия управления скоростью обучения примерно так и выглядит: начинаем с большой скорости, и постепенно эту скорость уменьшаем. Часто в этом случае используется 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 (например, число мини-батчей или число эпох), а T – отвечающий за скорость уменьшения η параметр. Впрочем, мы вряд ли сильно упростили себе задачу – теперь вместо одного параметра η нам надо подбирать два: η_0, T :) Но дело даже не в этом. Наверное, более глобально проблема заключается в следующем: такой подход никак не отражает характер и форму функции, минимум которой мы ищем. Например, если минимум находится в вытянутом «овраге», шаг спуска будет направлен от одной стенки оврага к другой. Когда же точка окажется на второй стенке, градиент опять будет направлен к противоположной стенке,

и так далее. В итоге, к минимуму мы будем двигаться очень и очень долго, рисунок 8. Попробуем исправить это недоразумение.

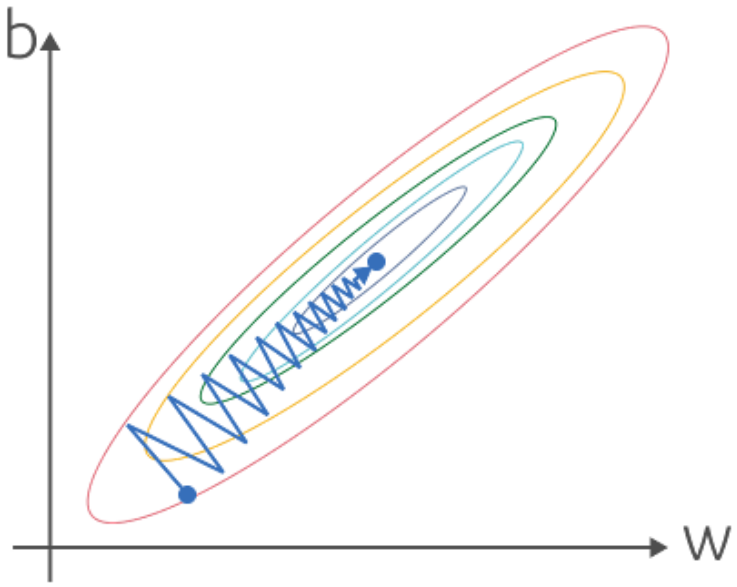

Рис. 8: Неудачная инициализация в овраге

2.3 Моментум (Momentum), импульс

Итак, метод импульсов (keras.optimizers.SGD(momentum=0.01, nesterov=True)) позволяет ускорить градиентный спуск в нужном (влекущем минимизацию) направлении и уменьшает его, спуска, колебания. С точки зрения физики все достаточно понятно: если считать, что в многомерном пространстве действуют те же законы ньютоновской механики, что и в нашей обыденности, то условный шарик, имеющий некоторую массу, будет быстро набирать скорость, скатываясь по крутому участку вниз к пологой части. Если же шарик попадет на плато, он все равно продолжит катиться по этому плато, и даже если уклон изменился на противоволожный, несколько итераций шарик все равно будет двигаться в ту же сторону, пока его не остановит трение. Метод моментов (импульсов) и моделирует подобную ситуацию. Формально шаг метода описывается следующим образом:

$$v_t = \gamma v_{t-1} + \eta \nabla L(\omega), \quad \omega = \omega - v_t,$$

где γ – параметр, меньший единицы (часто больший, чем 0.9), отвечающий за то, какую часть прошлого градиента мы готовы взять, чтобы скомбинировать его с текущим, L – функция потерь, которую мы и минимизируем.

Продолжая аналогию с катящимся с горы шариком, полезно не просто катиться вниз с бешеной скоростью, но и посматривать под ноги, чтобы случайно не сподкнуться. Это означает, что градиент разумно вычислять не в точке ω, а в точке $\omega - \gamma v_{t-1}$, то есть в точке, куда мы придем после применения момента предыдущего шага:

$$v_t = \gamma v_{t-1} - \eta \nabla_\omega L(\omega - \gamma v_{t-1}).$$

Перед нами – так называемый метод Нестерова.

2.4 Метод Ньютона

Следующая проблема градиентного спуска в многомерных пространствах – это седловые точки. Напомним, что точка называется седловой, если градиент в ней равен нулю, но точка – не точка минимума или максимума. Иными словами, по некоторым направлениям в этой точке функция минимизируется, а по некоторым – максимизируется, рисунок 9. Такие точки встречаются сплошь и рядом, но они нам совершенно неинтересны, и даже вредны, по понятным причинам. На наше счастье, есть несколько успешных модификаций градиентного спуска, которые позволяют немного отойти от седловой точки и успешно «скатиться вниз» по тем направлениям, которые максимизируют (или минимизируют) данную функцию в заданной точке.

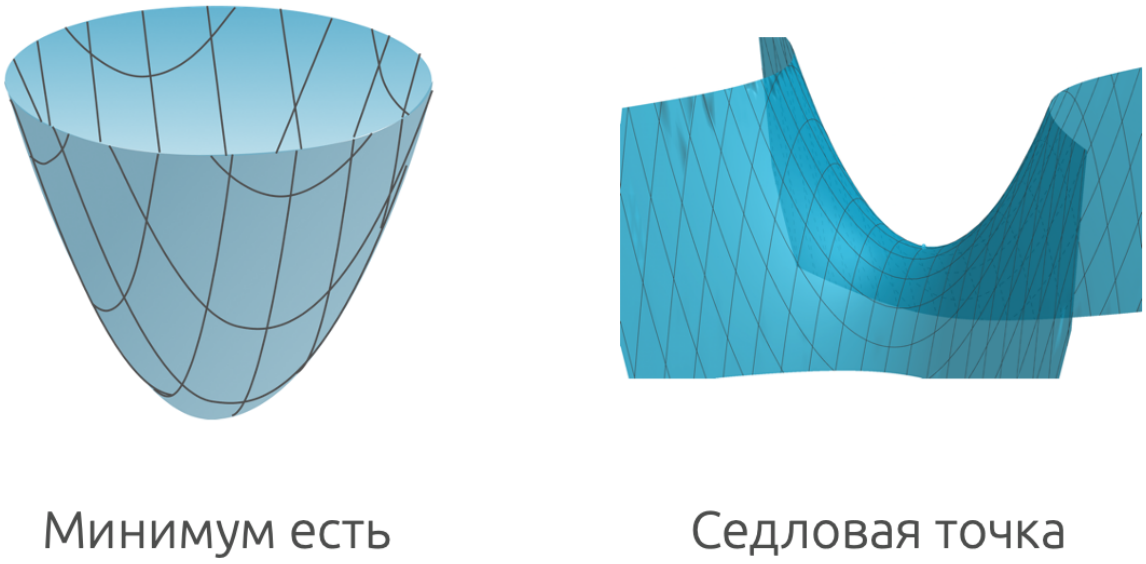

Рис. 9: Седловая точка

Градиент – это линейное приближение изменения функции, и оно не отслеживает (и не может отследить) форму функции в окрестности точки. А ведь форма скажет нам о многом: в окрестностях минимума и максимума

она чашечкообразная, а в окрестности седловой точки – совсем нет. Одно из возможных решений, помогающих следить за формой функции – это искать не первую производную, а вторую – так называемый метод Ньютона. В основе метода Ньютона лежит идея приближения функции квадратичной функцией (по сути – формула Тейлора второго порядка):

$$L(\omega) \approx L(\omega_0) + \nabla_\omega L(\omega_0)(\omega - \omega_0) + \frac{1}{2}(\omega - \omega_0)^T H(\omega)(\omega - \omega_0),$$

где $H(\omega)$ – гессиан $L(\omega)$. Этот метод, будучи методом второго порядка, требует очень много вычислений и памяти, поэтому в теории нейронных сетей используется редко. Мы сейчас поговорим с вами о его неплохих рабочих аналогах, используемых на практике.

2.5 Adagrad, Adadelta, RMSProp, Adam

Концепция метода Adagrad (adaptive gradient, keras.optimizers.adagrad) предлагает использовать не один и тот же константный learning rate для всех измерений на конкретном шаге, а менять его адаптивно для разных измерений. Это соображение не лишено логики. Предположим, что вам встретилась какая-то редкая хорошо оплачиваемая профессия, или какое-то причудливое слово в базе спам-фильтра – такие факты просто потонут в шуме всех остальных типичных обновлений. А значит, неплохо бы корректировать параметры с оглядкой на типичность фиксируемого признака. Например, давайте хранить для каждого параметра сумму квадратов его обновлений. Если параметр часто обновляется, то его дергают туда-сюда, и сумма быстро накапливается, и наоборот.

Пусть

$$g_{t,i} = \nabla_{\omega_i} L(\omega)$$

– градиент функции ошибки по параметру ω_i на шаге t. Тогда обновление параметра ω_i будем производить по следующему правилу:

$$\omega_{t+1,i} = \omega_{t,i} - \frac{\eta}{\sqrt{G_{t,ii} + \varepsilon}} g_{t,i},$$

где G_t – диагональная матрица, каждый диагональный элемент которой – сумма квадратов градиентов по соответствующему параметру за все предыдущие шаги, то есть

$$G_{t,ii} = G_{t-1,ii} + g_{t,i}^2,$$

а параметр $\varepsilon > 0$ (обычно порядка $10^{-6} - 10^{-8}$) помогает избежать деления на ноль. В итоге, идейно, мы искусственно повышаем интерес к направлениям, в которых изменение мало, и искусственно понижаем интерес к направлениям,

в которых изменение велико. Как результат, мы пытаемся как-то выравнять интерес ко всем направлениям.

Интересно заметить, что если убрать корень из знаменателя, то алгоритм будет работать куда хуже.

Итак, плюс заключается в том, что теперь learning rate обновляется автоматически на каждом шаге, и об изменении шага со временем обучения можно сильно не задумываться. Есть, конечно, и минус. Скорость обучения порой уменьшается слишком стремительно (ведь диагональные элементы матрицы G постоянно увеличиваются), а это плохо сказывается на обучении глубоких нейронных сетей. Методы Adadelta и RMSProp пытаются как раз-таки бороться с этим недочетом.

RMSProp (keras.optimizers.rmsprop) – root mean squares – использует усредненный по истории квадрта градиента. По сути, этот метод не накапливает сумму градиентов, как это делает Adagrad, а вычисляет что-то вроде скользящего среднего этих градиентов. Тем самым, в некотором смысле запрещается неограниченный рост суммы градиентов, и, как следствие, неограниченное падение learning rate.

Adadelta (keras.optimizers.adadelta) является еще одной модификацией RMSProp, мы не будем подробно останавливаться на деталях.

Последний метод, о котором мы поговорим – это так называемый Adam (keras.optimizers.adam). В каком-то смысле можно считать, что этот метод – симбиоз методов RMSProp и Momentum. У него все также есть адаптивный learning rate, но теперь адаптация применяется не просто к градиенту, а, в некотором смысле, к скорости. Данный алгоритм на данный момент является одним из самых популярных. Оптимизатор Adam аппроксимирует вторую производную в точке, что приближает его к методам, вычисляющим вторую производную, например к методу Ньютона. Это наделяет данный метод и некоторыми достоинствами, и недостатками. Например, часто требуется меньшее число итераций для достижения минимума или максимума, но большая вычислительная сложность.

Сравнение методов и скорости их работы можно хорошо увидеть на анимации. Прекрасно видно, как законы инерции работают, например, в случае метода моментов и метода Нестерова. И как Adam использует смесь RMSProp и метода моментов.

2.6 Практические советы

Естественно, такое разнообразие алгоритмов (и описаны они далеко не все) приводит к естественному вопросу: а какой же алгоритм выбрать для конкретной задачи? Этот вопрос не имеет какого-то глобального ответа, на практике нужно пробовать разные варианты. Впрочем, если данные достаточно разрежены, то адаптивные алгоритмы неплохо справляются со своей

задачей.

Важно отметить и следующее замечание. На практике при использовании спуска важно отслеживать ошибку на валидационном множестве и останавливать процесс в тот момент, когда ошибка начинает расти. Иначе – не избежать оверфиттинга, ведь можно найти достаточно глубокий минимум функции ошибки, который плохо обобщается на новые данные (плохая генерализующая способность). Это и понятно: алгоритмы минимизации не в курсе, что там с обобщением. Они заточены под одну задачу: минимизируй, минимизируй, минимизируй. Так что их порывы неплохо бы контролировать.

Кстати, приведенные рассуждения еще раз выступают в пользу деления набора данных на три части: тренировочную, валидационную и тестовую. Ведь, если использовать описанный философский подход, валидационное множество уже используется для обучения модели и, в некотором смысле, превращается в тренировочное.