Приложение № 2
к распоряжению РТУ МИРЭА
от__ _____№_____

Титульный лист материалов по дисциплине
(заполняется по каждому виду учебного материала)

ДИСЦИЛИНА Проектирование и обучение нейронных сетей
(полное наименование дисциплины без сокращений)

ИНСТИТУТ Информационые технологии

КАФЕДРА Вычислительная техника
полное наименование кафедры

ВИД УЧЕБНОГО МАТЕРИАЛА Лекция
(в соответствии с пп.1-11)

ПРЕПОДАВАТЕЛЬ Сорокин Алексей Борисович
(фамилия, имя, отчество)

СЕМЕСТР 7 семестр, 2023/2024
(указать семестр обучения, учебный год)

15. ЛЕКЦИЯ. Оптимизация в обучении глубоких моделей

Глубокие сети прямого распространения, которые называют также нейронными сетями прямого распространения, или многослойными перцептронами – самые типичные примеры моделей глубокого обучения.

Алгоритмы глубокого обучения включают оптимизацию в самых разных контекстах (глубокое обучение – это вид машинного обучения, наделяющий компьютеры способностью учиться на опыте и понимать мир в терминах иерархии концепций). В этой связи нас будет интересовать один частный случай: нахождение параметров θ нейронной сети, значительно уменьшающих функцию стоимости $J(\theta)$, которая обычно служит мерой качества, вычисляется на всем обучающем наборе и содержит дополнительные регуляризирующие члены (функция стоимости – это мера того, насколько модель ошибочна с точки зрения ее способности оценивать отношения между X и y. Обычно это выражается как разница или расстояние между прогнозируемым значением и фактическим значением).

Зададимся вопросом, чем оптимизация, используемая в алгоритме машинного обучения, отличается от чистой оптимизации.

Чем обучение отличается от чистой оптимизации

Алгоритмы оптимизации, используемые для обучения глубоких моделей, отличаются от традиционных алгоритмов оптимизации в нескольких отношениях. Машинное обучение обычно работает не напрямую. В большинстве ситуаций нас интересует не которая мера качества P, которая определена относительно тестового набора и может оказаться вычислительно неприступной. Поэтому мы оптимизируем P косвенно. Мы уменьшаем другую функцию стоимости *J(θ)* в надежде, что при этом улучшится и P. Это резко отличается от чистой оптимизации, где минимизация J и есть конечная цель. Кроме того, алгоритмы оптимизации для обучения глубоких моделей обычно включают специализации для конкретной структуры целевых функций.

Типичную функцию стоимости можно представить в виде среднего по обучающему набору:

$$J(\theta) = \mathbb{E}_{x,y\sim\hat{p}_{data}} L(f(x;\theta), y)$$

где L – функция потерь на одном примере, $f(x;\theta)$ – предсказанный выход для входа x, а $\hat{p}_{data}$ – эмпирическое распределение. В случае обучения с учителем y – ассоциированная с входом метка. Аргументами L являются $f(x;\ \theta)$ и y – это нерегуляризированное обучение с учителем (то есть без добавления некоторых дополнительных ограничений к условию с целью решить

некорректно поставленную задачу или предотвратить переобучение). Этот случай тривиально обобщается, например, на включение в качестве аргументов θ или x или на исключение y из числа аргументов с целью разработки различных видов регуляризации или обучения без учителя.

Это уравнение определяет целевую функцию относительно обучающего набора. Но обычно предпочитается минимизировать соответствующую целевую функцию, в которой математическое ожидание берется по порождающему данные распределению p_{data}, а не просто по конечному обучающему набору:

$$J^*(\theta) = \mathbb{E}_{x,y\sim p_{data}} L(f(x;\theta), y)$$

Цель алгоритма машинного обучения – уменьшить математическое ожидание ошибки обобщения, описываемое последней формулой. Эта величина называется риском. Подчеркну еще раз, что математическое ожидание берется по истинному распределению p_{data}. Если бы мы знали истинное распределение $p_{data}(x, y)$, то минимизация риска была бы задачей оптимизации, решаемой с помощью алгоритма оптимизации. Но когда истинное распределение $p_{data}(x, y)$ неизвестно, а есть только обучающий набор примеров, мы имеем задачу машинного обучения. Простейший способ преобразовать задачу машинного обучения в задачу оптимизации – минимизировать ожидаемые потери на обучающем наборе. Это значит, что мы заменяем истинное распределение эмпирическим распределением $\hat{p}_{data}$, определяемым по обучающему набору (выборочные данные, полученные в ходе эксперимента, называются эмпирическими данными и на основе этих данных можно построить эмпирическое распределение, то есть распределение элементов выборки по значениям изучаемого признака).

Процесс обучения, основанный на минимизации этой средней ошибки обучения, называется минимизацией эмпирического риска. В такой постановке машинное обучение все еще очень похоже на чистую оптимизацию. Вместо оптимизации риска напрямую мы оптимизируем эмпирический риск и надеемся, что и риск тоже заметно уменьшится. Существуют теоретические результаты, устанавливающие условия, при которых можно ожидать того или иного уменьшения истинного риска.

Тем не менее минимизация эмпирического риска уязвима для переобучения. Модели высокой емкости могут попросту запомнить обучающий набор. Во многих случаях минимизация эмпирического риска практически неосуществима. Самые эффективные современные алгоритмы оптимизации основаны на градиентном спуске, но для многих полезных функций потерь, например бинарной, производная малоинтересна (либо равна нулю, либо не

определена). Из-за этих двух проблем минимизация эмпирического риска редко применяется в контексте глубокого обучения. Вместе нее используется несколько иной подход, при котором фактически оптимизируемая величина еще сильнее отличается от той, которую мы хотели бы оптимизировать на самом деле.

Иногда реально интересующая нас функция потерь (скажем, ошибка классификации) и та, что может быть эффективно оптимизирована, – «две большие разницы». Например, задача точной минимизации ожидаемой бинарной функции (булева функция) потерь обычно неразрешима (она экспоненциально зависит от размерности входных данных). В таких случаях оптимизируют суррогатную функцию потерь, выступающую в роли заместителя истинной, но обладающую рядом преимуществ (например, в качестве суррогата бинарной функции потерь часто берут отрицательное логарифмическое правдоподобие правильного класса). В некоторых случаях суррогатная функция потерь позволяет достичь даже больших успехов в обучении. Поэтому очень важное различие между оптимизацией вообще и применяемой в алгоритмах обучения заключается в том, что алгоритмы обучения обычно останавливаются не в локальном минимуме. Вместо этого алгоритм, как правило, минимизирует суррогатную функцию потерь, но останавливается, когда выполнено условие сходимости, основанное на идее ранней остановки. Типичное условие ранней остановки основано на истинной функции потерь, например вычислении бинарной функции потерь на контрольном наборе, и предназначено для того, чтобы остановить работу алгоритма, когда возникает угроза переобучения. Обучение зачастую заканчивается, когда производные суррогатной функции потерь все еще велики, и этим разительно отличается от чистой оптимизации, при которой считается, что алгоритм сошелся, если градиент стал очень малым.

Пакетные и мини-пакетные алгоритмы

Еще одно отличие алгоритмов машинного обучения от общих алгоритмов оптимизации состоит в том, что целевая функция обычно представлена в виде суммы по обучающим примерам. Типичный алгоритм оптимизации в машинном обучении вычисляет каждое обновление параметров, исходя из ожидаемого значения функции стоимости, оцениваемого только по подмножеству членов полной функции стоимости.

На практике можно случайно выбрать небольшое число примеров и усреднить только по ним. Напомним, что стандартная ошибка среднего, оцененная по выборке объема n, равна $\sigma/\sqrt{n}$, где σ – истинное стандартное отклонение выборки. Знаменатель показывает, что точность оценки градиента с

увеличением объема выборки растет медленнее. Сравним две гипотетические оценки градиента, одна на основе 100 примеров, другая – 10 000. Для вычисления второй оценки потребуется в 100 раз больше времени, но стандартная ошибка среднего уменьшится только в 10 раз. Большинство алгоритмов оптимизации сходится гораздо быстрее, если им позволено быстро вычислять приближенные оценки градиента вместо медленного вычисления точного значения.

Еще одно сообщение в пользу статистического оценивания градиента по небольшой выборке связано с избыточностью обучающего набора. В худшем случае все m примеров в обучающем наборе в точности совпадают. Оценка градиента по выборке дала бы правильное значение, взяв всего один пример, т. е. было бы затрачено в m раз меньше времени, чем при наивном подходе. На практике нам вряд ли встретится худший случай, но все же можно найти много примеров, дающих очень похожий вклад в градиент. Алгоритмы оптимизации, в которых используется весь обучающий пакет, называются пакетными, или детерминированными, градиентными методами, поскольку обрабатывают сразу все примеры одним большим пакетом. Эта терминология может вызывать путаницу, потому что слово «пакет» часто употребляется также для обозначения мини-пакета, применяемого в алгоритме стохастического градиентного спуска.

Алгоритмы оптимизации, в которых используется по одному примеру за раз, иногда называют стохастическими, или онлайновыми, методами. Термин «онлайновый» обычно резервируется для случая, когда примеры выбираются из непрерывного потока, а не из обучающего набора фиксированного размера, по которому можно совершать несколько проходов.

Большинство алгоритмов, используемых в глубоком обучении, находится где-то посередине – число примеров в них больше одного, но меньше размера обучающего набора. Традиционно они назывались мини-пакетными стохастическими, методами, а сейчас – просто стохастическими. Канонический пример стохастического метода – стохастический градиентный спуск, который будет нами рассмотрен.

На размер мини-пакета оказывают влияние следующие факторы:

- чем больше пакет, тем точнее оценка градиента, но зависимость хуже линейной;
- если пакет очень мал, то не удается в полной мере задействовать преимущества многоядерной архитектуры. Поэтому существует некий абсолютный минимум размера пакета – такой, что обработка мини-пакетов меньшего размера не дает никакого выигрыша во времени;
- если все примеры из пакета нужно обрабатывать параллельно (так

обычно и бывает), то размер пакета лимитирован объемом памяти. Для многих аппаратных конфигураций размер пакета – ограничивающий фактор;

- для некоторых видов оборудования оптимальное время выполнения достигается при определенных размерах массива. Типичный пакет имеет размер от 32 до 256, а для особо больших моделей иногда пробуют 16;

В зависимости от вида алгоритма используется разная информация из мини-пакета, причем разными способами. Методы, которые вычисляют обновления только на основе градиента g, обычно сравнительно устойчивы и могут работать с пакетами небольшого размера, порядка 100. Методы второго порядка, в которых используется также матрица Гессе H и которые вычисляют такие обновления, как $H^{-1}g$, обычно нуждаются в пакетах гораздо большего размера, порядка 10 000. Такие большие пакеты нужны, чтобы свести к минимуму флуктуации в оценках $H^{-1}g$.

Важно также, чтобы мини-пакеты выбирались случайно. Для вычисления несмещенной оценки ожидаемого градиента по выборке необходимо, чтобы примеры были независимы. Таким образом, когда порядок примеров в наборе не случаен, необходимо перетасовать пакет, прежде чем формировать мини-пакеты. На практике обычно достаточно перетасовать набор один раз и затем хранить его в таком виде. Такое отклонение от истинно случайного выбора не оказывает значимого негативного эффекта. Тогда как полное пренебрежение перетасовкой примеров способно серьезно снизить эффективность алгоритма.

Проблемы оптимизации нейронных сетей

В общем случае оптимизация – чрезвычайно трудная задача. Традиционно в машинном обучении избегали сложностей общей оптимизации за счет тщательного выбора целевой функции и ограничений, гарантирующих выпуклость задачи оптимизации. При обучении нейронных сетей приходится сталкиваться с общим невыпуклым случаем. Но даже выпуклая оптимизация не обходится без осложнений. Отметим нескольких наиболее заметных проблем оптимизации, возникающих в процессе обучения глубоких моделей.

- Плохая обусловленность

Ряд проблем возникает даже при оптимизации выпуклых функций. Самая известная из них – плохая обусловленность матрицы Гессе H.

В многомерном случае в одной точке вторые производные по каждому направлению различны. Число обусловленности матрицы Гессе в точке измеряет степень различия вторых производных. Если число обусловленности велико, то градиентный спуск будет работать плохо. Это объясняется тем, что в одном направлении производная растет быстро, а в другом медленно. Метод

градиентного спуска не в курсе этого различия, поэтому не знает, что предпочтительным направлением для исследования является то, в котором производная дольше остается отрицательной. Из-за плохого числа обусловленности трудно выбрать хорошую величину шага. Шаг должен быть достаточно малым, что не пропустить минимум и подниматься вверх во всех направлениях, где кривизна строго положительна. Но обычно это означает, что шаг слишком мал для заметного продвижения в направлениях с меньшей кривизной. Пример приведен на рис. 1.

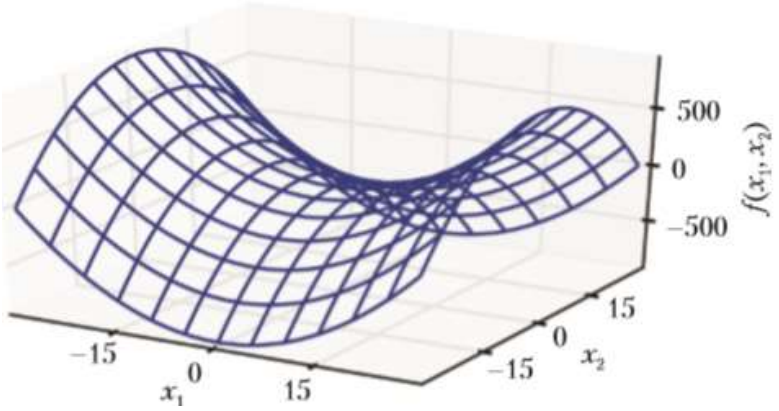

Рис. 1. График функции $f(x) = x_1^2 - x_2^2$

Вдоль оси x_1 функция изгибается вверх. Направление этой оси – собственный вектор матрицы Гессе с положительным собственным значением. Вдоль оси x_2 функция изгибается вниз. Это направление собственного вектора матрицы Гессе с отрицательным собственным значением. Таким образом в «седловой точке» присутствует как положительная, так и отрицательная кривизна. Тогда собственные значения разных знаков, в одном сечении достигается локальный максимум, а в другом – локальный минимум

Хотя плохая обусловленность характерна не только для обучения нейронных сетей, некоторые методы борьбы с ней, используемые в других контекстах, к нейронным сетям плохо применимы. Например, метод Ньютона – отличное средство минимизации выпуклых функций с плохо обусловленными гессианами (определителями), метод нуждается в существенной модификации.

- Локальные минимумы

Одна из самых важных черт выпуклой оптимизации состоит в том, что такую задачу можно свести к задаче нахождения локального минимума. Гарантируется, что любой локальный минимум одновременно является глобальным. У некоторых выпуклых функций в нижней части графика имеется не единственный глобальный минимум, а целый плоский участок. Однако любая точка на плоском участке является допустимым решением. При оптимизации выпуклой функции точно устанавливается что, обнаружив критическую точку

любого вида, находится хорошее решение.

У невыпуклых функций, в частности нейронных сетей, локальных минимумов может быть несколько. Более того, почти у любой глубокой модели гарантированно имеется множество локальных минимумов. Впрочем, это не всегда является серьезной проблемой. Локальные минимумы становятся проблемой, если значение функции стоимости в них велико, по сравнению со значением в глобальном минимуме.

- Плато, седловые точки и другие плоские участки

Для многих невыпуклых функций в многомерных пространствах локальные минимумы (и максимумы) встречаются гораздо реже других точек с нулевым градиентом: седловых точек. В одних точках в окрестности седловой стоимость выше, чем в седловой точке, в других – ниже. В седловой точке матрица Гессе имеет как положительные, так и отрицательные собственные значения. В точках, лежащих вдоль собственных векторов с положительными собственными значениями, стоимость выше, чем в седловой точке, а в точках, лежащих вдоль собственных векторов с отрицательными собственными значениями, – ниже. Можно считать, что седловая точка является локальным минимумом в одном сечении графика функции стоимости и локальным максимумом – в другом. Это иллюстрирует рис. 1.

Для метода Ньютона седловые точки представляют очевидную проблему. Идея алгоритма градиентного спуска – «спуск с горы», а не явный поиск критических точек. С другой стороны, метод Ньютона специально предназначен для поиска точек с нулевым градиентом. Без надлежащей модификации он вполне может найти седловую точку. Изобилие седловых точек в многомерных пространствах объясняет, почему методы второго порядка не смогли заменить градиентный спуск в обучении нейронных сетей. Методы второго порядка все еще с трудом масштабируются на большие нейронные сети, но если этот бесседловой метод удастся масштабировать, то он сулит интересные перспективы.

Могут также существовать широкие плоские области с постоянным значением. В этих областях равны нулю и градиент, и гессиан. Такие вырожденные участки – серьезная проблема для всех алгоритмов численной оптимизации. В выпуклой задаче широкая плоская область должна целиком состоять из глобальных минимумов, но в общем случае ей могут соответствовать и большие значения целевой функции.

- Утесы и резко растущие градиенты

В нейронных сетях с большим числом слоев часто встречаются очень

крутые участки, напоминающие утесы (рис. 2). Это связано с перемножением нескольких больших весов. На стене особенно крутого утеса шаг обновления градиента может привести к очень сильному изменению параметров, что обычно заканчивается «срывом» с утеса. Утес представляет опасность вне зависимости от того, приближаемся к нему сверху или снизу, но, по счастью, самых печальных последствий можно избежать с помощью эвристической техники отсечения градиента.

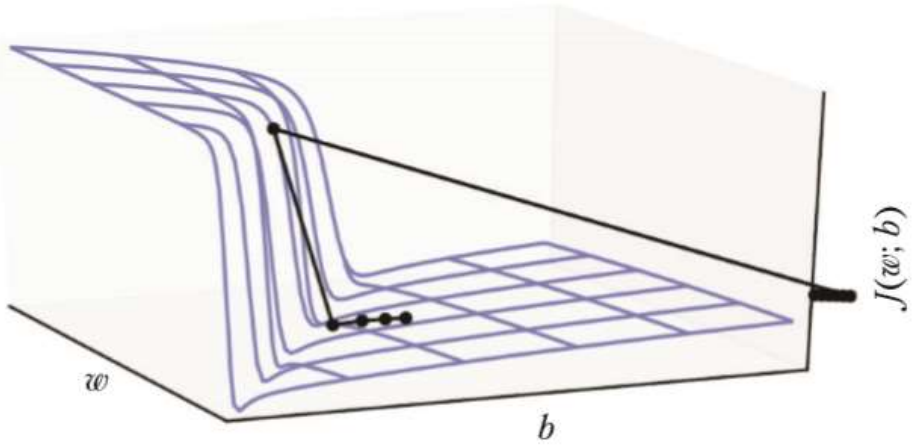

Рис. 2. Утес

Основная идея – вспомнить, что градиент определяет не оптимальный размер шага, а лишь оптимальное направление в бесконечно малой области. Когда традиционный алгоритм градиентного спуска предлагает сделать очень большой шаг, вмешивается эвристика отсечения, и в результате шаг уменьшается, так что становится менее вероятным выход за пределы области, в которой градиент указывает приближенное направление самого крутого спуска. Утесы чаще всего встречаются в функциях стоимости рекуррентных нейронных сетей, поскольку в таких моделях вычисляется произведение большого числа множителей, по одному на каждый временной шаг. Поэтому чем длиннее временная последовательность, тем больше количество перемножений.

- Неточные градиенты

Большинство алгоритмов оптимизации исходит из предположения, что имеется доступ к точному градиенту или гессиану. На практике же обычно налицо только зашумленная или даже смещенная оценка этих величин. Почти все алгоритмы глубокого обучения опираются на выборочные оценки, по крайней мере в том, что касается использования мини-пакета обучающих примеров для вычисления градиента. Бывает и так, что целевая функция, которую мы хотим минимизировать, вычислительно неразрешима. В таком случае неразрешимой обычно является и задача вычисления градиента, и тогда нам остается только аппроксимировать градиент. Проблему можно обойти путем

выбора суррогатной функции потерь.

- Плохое соответствие между локальной и глобальной структурами

Многие рассмотренные выше проблемы касаются свойств функции потерь в одной точке – трудно сделать следующий шаг, если $J(\theta)$ плохо обусловлена в текущей точке θ, или если θ находится на стене утеса, или если θ является седловой точкой, маскирующей возможность добиться улучшения путем «спуска с горы». Все эти проблемы можно преодолеть в одной точке и тем не менее остаться «на бобах», если найденное направление наибольшего локального улучшения не ведет в сторону отдаленных областей с гораздо меньшей стоимостью.

На рис. 3 видно, что траектория обучения резко удлиняется из-за необходимости обогнуть по широкой дуге скалообразную структуру.

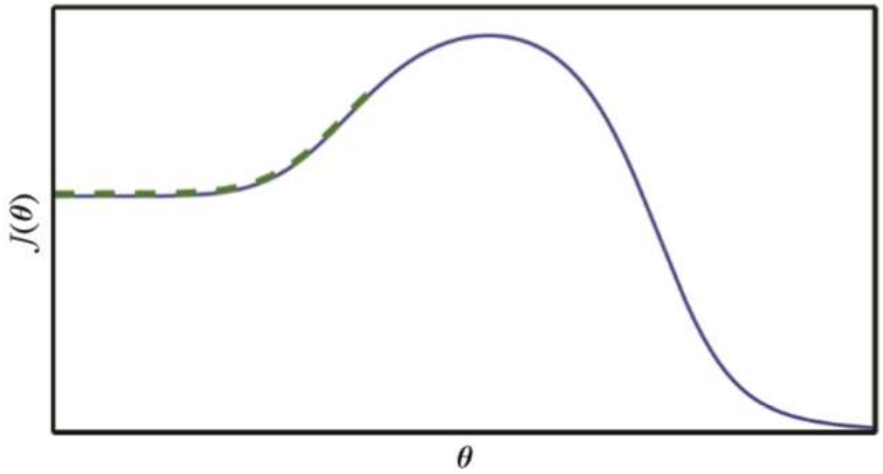

Рис. 3. Траектория обучения

Оптимизация, основанная на локальном спуске, может потерпеть неудачу (рис.3), если локальное направление не ведет к глобальному решению. Здесь показано, как такое может произойти даже в отсутствие локальных минимумов или седловых точек. Функция стоимости в этом примере только асимптотически приближается к низким значениям, но не имеет минимумов. Проблема вызвана тем, что начальные значения выбраны не по ту сторону «горы», и алгоритм не может перебраться через нее. В многомерных пространствах алгоритмы обучения обычно способны обогнуть такие горы, но траектория может оказаться длинной, и обучение займет слишком много времени.

Основные алгоритмы

Мы уже познакомились с алгоритмом градиентного спуска, идея которого – перемещаться в направлении убывания градиента всего обучающего набора. Работу можно значительно ускорить, воспользовавшись стохастическим градиентным спуском для случайно выбранных мини-пакетов.

Стохастический градиентный спуск

Метод стохастического градиентного спуска (СГС) и его варианты – пожалуй, самые употребительные алгоритмы машинного обучения вообще и глубокого обучения в частности. Таким образом, можно получить несмещенную оценку градиента, усреднив его по мини-пакету m независимых и одинаково распределенных примеров, выбранных из порождающего распределения. В алгоритме 1 показано, как осуществить спуск вниз, пользуясь этой оценкой.

Алгоритм 1. Обновление на k-ой итерации стохастического градиентного спуска.

Require: скорость обучения ε_k

Require: Начальные значения параметров θ

while критерий остановки не выполнен **do**

Выбрать из обучающего набора мини-пакет m примеров $\{x^{(1)}, \dots, x(m)\}$ и соответствующие им метки $y^{(i)}$.

Вычислить оценку градиента: $g \leftarrow +(1/m)\nabla_\theta \Sigma_i L(f(x^{(i)};\, \theta), y^{(i)})$

Применить обновление: $\theta \leftarrow \theta - \varepsilon\hat{g}$.

end while

Require – требуется, while – цикл пока, do – делать.

Основной параметр алгоритма СГС – скорость обучения, которую необходимо постепенно уменьшать ее со временем, поэтому обозначается ε_k скорость обучения на k-ой итерации. Связано это с тем, что оценка градиента вносит источник шума (случайная выборка m обучающих примеров), который не исчезает, даже когда нашли минимум. Напротив, при использовании пакетного градиентного спуска истинный градиент полной функции стоимости уменьшается по мере приближения к минимуму и обращается в 0 в самой точке минимума, так что скорость обучения можно зафиксировать. Достаточные условия сходимости СГС имеют вид:

$$\sum_{k=1}^{\infty} \varepsilon_k = \infty \text{ и } \sum_{k=1}^{\infty} \varepsilon_k^2 < \infty$$

На практике скорость обучения обычно уменьшают линейно до итерации с номером τ:

$$\varepsilon_k = (1-\alpha)\varepsilon_0 + \alpha\varepsilon_\tau$$

где $\alpha = k/\tau$. После τ-й итерации ε остается постоянным.

Скорость обучения можно выбрать методом проб и ошибок, но обычно лучше понаблюдать за кривыми обучения – зависимостью целевой функции от времени. Если скорость изменяется линейно, то нужно задать параметры ε_0, ε_τ и τ. Обычно в качестве τ выбирают число итераций, необходимое для выполнения нескольких сотен проходов по обучающему набору. Величину ε_τ задают равной

примерно 1% от ε_0. Главный вопрос: как задать ε_0. Если значение слишком велико, то кривая обучения будет сильно осциллировать (клебаться), а функция стоимости – значительно увеличиваться. Слабые осцилляции не несут угрозы, особенно если для обучения используется стохастическая функция стоимости. Если скорость обучения слишком мала, то обучение происходит медленно, а если слишком мала и начальная скорость, то обучение может застрять в точке с высокой стоимостью. Как правило, оптимальная начальная скорость обучения с точки зрения общего времени обучения и конечной стоимости выше, чем скорость, которая дает наилучшее качество после первых примерно 100 итераций. Поэтому обычно имеет смысл последить за первыми несколькими итерациями и взять скорость обучения большую, чем наилучшая на этом отрезке, но не настолько высокую, чтобы дело закончилось сильной неустойчивостью. Самое важное свойство СГС и схожих методов мини-пакетной или онлайновой градиентной оптимизации заключается в том, что время вычислений в расчете на одно обновление не увеличивается с ростом числа обучающих примеров. Следовательно, сходимость возможна, даже когда число обучающих примеров очень велико. Если набор данных достаточно велик, то СГС может сойтись с некоторым фиксированным отклонением от финальной ошибки на тестовом наборе еще до завершения обработки всего обучающего набора. Для изучения скорости сходимости алгоритма оптимизации часто измеряют **ошибку превышения** $\boldsymbol{J(\theta)} - min_\theta J(\theta)$, т. е. величину, на которую текущая функция стоимости превышает минимально возможную стоимость. При применении СГС к выпуклой задаче ошибка превышения равна $O(1/\sqrt{k})$ после k итераций. Тогда как в задачах машинного обучения не имеет смысла искать алгоритм оптимизации, который сходится быстрее, чем O(1/k), – более быстрая сходимость, скорее всего, приведет к переобучению. Кроме того, асимптотический анализ (т.е. метод описания предельного поведения функций) игнорирует многие преимущества стохастического градиентного спуска с небольшим числом шагов. На больших наборах способность СГС быстро достигать прогресса на начальной стадии после обсчета небольшого числа примеров перевешивает его медленную асимптотическую сходимость. Иногда можно также попытаться найти компромисс между пакетным и стохастическим градиентным спусками, постепенно увеличивая размер мини-пакета в процессе обучения.

Импульсный метод

Стохастический градиентный спуск остается популярной стратегией оптимизации, но обучение с его помощью иногда происходит слишком

медленно. Импульсный метод призван ускорить обучение, особенно в условиях высокой кривизны, небольших, но устойчивых градиентов или зашумленных градиентов. В импульсном алгоритме вычисляется экспоненциально затухающее скользящее среднее прошлых градиентов и продолжается движение в этом направлении. Работа импульсного метода иллюстрируется на рис. 1.

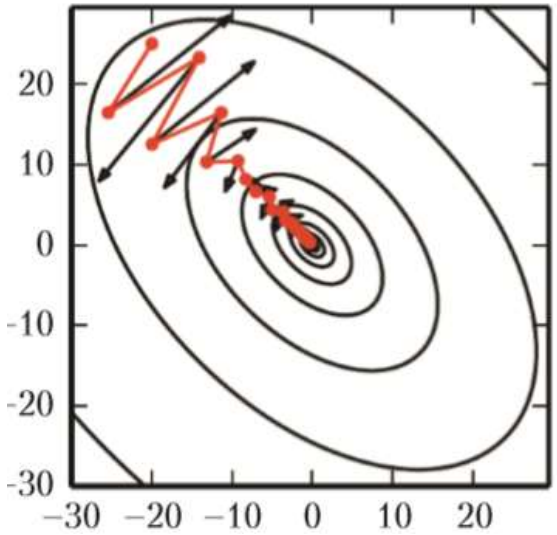

Рис.1. Работа импульсного метода

Импульсный метод призван решить две проблемы: плохую обусловленность матрицы Гессе и дисперсию стохастического градиента. На рис. 1 показано, как преодолевается первая проблема. Эллипсы обозначают изолинии квадратичной функции потерь с плохо обусловленной матрицей Гессе. Красная линия, пересекающая эллипсы, соответствует траектории, выбираемой в соответствии с правилом обучения методом моментов в процессе минимизации этой функции. Для каждого шага обучения стрелка показывает, какое направление выбрал бы в этот момент метод градиентного спуска. Как видно, плохо обусловленная квадратичная целевая функция выглядит как длинная узкая долина или овраг с крутыми склонами. Импульсный метод правильно перемещается вдоль оврага, тогда как градиентный спуск впустую тратил бы время на перемещение вперед-назад поперек оврага. Сравните также с рис. 2, где показано поведение градиентного спуска без учета импульс. Здесь число обусловленности матрицы Гессе в точке измеряет степень различия вторых производных. Если число обусловленности велико, то градиентный спуск будет работать плохо. Это объясняется тем, что в одном направлении производная растет быстро, а в другом медленно. Метод градиентного спуска не в курсе этого различия, поэтому не знает, что предпочтительным направлением для исследования является то, в котором производная дольше остается отрицательной. Из-за плохого числа обусловленности трудно выбрать хорошую

величину шага. Шаг должен быть достаточно малым, что не пропустить минимум и подниматься вверх во всех направлениях, где кривизна строго положительна. Но обычно это означает, что шаг слишком мал для заметного продвижения в направлениях с меньшей кривизной.

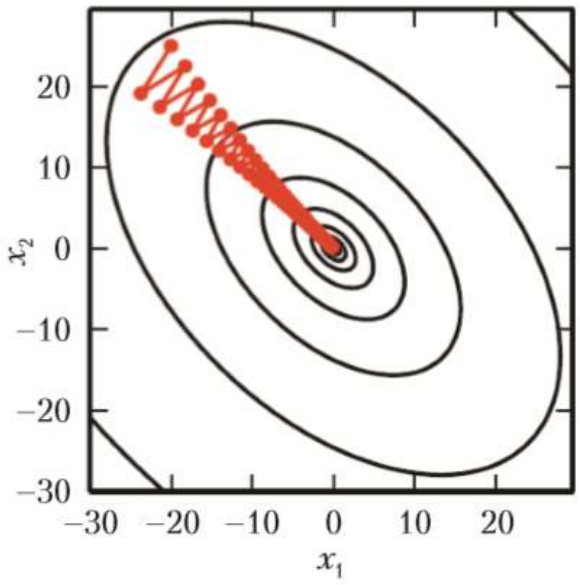

Рис.2. Работа градиентного метода

Формально говоря, в импульсном алгоритме вводится переменная v, играющая роль скорости, – это направление и скорость перемещения в пространстве параметров. Скорость устанавливается равной экспоненциально затухающему скользящему среднему градиента со знаком минус. Название алгоритма проистекает из физической аналогии, согласно которой отрицательный градиент – это сила, под действием которой частица перемещается в пространстве параметров согласно законам Ньютона. В физике импульсом называется произведение массы на скорость. В импульсном алгоритме масса предполагается единичной, поэтому вектор скорости v можно рассматривать как импульс частицы. Гиперпараметр $\alpha \in [0,1)$ определяет скорость экспоненциального затухания вкладов предшествующих градиентов (Гиперпараметры модели — параметры, значения которых задается до начала обучения модели и не изменяется в процессе обучения). Правило обновления имеет вид:

$$v \leftarrow \alpha v \leftarrow \varepsilon \nabla_\theta \left(\frac{1}{m} \sum_{i=1}^{m} L\left(f\left(x^{(i)}; \theta\right), y^{(i)}\right) \right)$$

$$\theta \leftarrow \theta + v$$

Алгоритм 2. Стохастический градиентный спуск (СГС) с учетом импульса

Require: скорость обучения ε, параметр импульса α

Require: начальные значения параметров θ, начальная скорость v

while критерий остановки не выполнен **do**

Выбрать из обучающего набора мини-пакет m примеров $\{x^{(1)}, \ldots, x(m)\}$ и соответствующие им метки $y^{(i)}$.

Вычислить оценку градиента: $g \leftarrow (1/m)\nabla_\theta \Sigma_i L(f(x^{(i)};\, \theta), y^{(i)})$

Вычислить обновление скорости: $v \leftarrow \alpha v - \varepsilon g$.

Применить обновление: $\theta \leftarrow \theta + v$.

end while

В скорости v суммируются градиенты $\nabla_\theta((1/m)\Sigma_{i=1}^{m} L(f(x^{(i)};\, \theta),\, y^{(i)}))$. Чем больше α относительно ε, тем сильнее предшествующие градиенты влияют на выбор текущего направления. СГС с учетом импульса описан в алгоритме 2. Шаг зависит от величины и сонаправленности предшествующих градиентов. Размер шага максимален, когда много последовательных градиентов указывают точно в одном и том же направлении. Если импульсный алгоритм всегда видит градиент g, то он будет ускоряться в направлении $-g$, пока не достигнет конечной скорости, при которой размер шага равен

$$\frac{\varepsilon \|g\|}{1-\alpha}$$

Таким образом, полезно рассматривать гиперпараметр импульса в терминах $1/(1-\alpha)$. Как и скорость обучения, α может меняться со временем. Как правило, начинают с небольшого значения и постепенно увеличивают его. Изменение α со временем не так важно, как уменьшение ε со временем.

Импульсный алгоритм можно рассматривать как имитацию движения частицы, подчиняющейся динамике Ньютона. Физическая аналогия помогает составить интуитивное представление о поведении алгоритма градиентного спуска и импульсного метода. Положение частицы в любой момент времени описывается функцией $\theta(t)$. К частице приложена суммарная сила $f(t)$, под действием которой частица ускоряется:

$$f(t) = \frac{\partial^2}{\partial t^2}\,\theta(t)$$

Вместо того чтобы рассматривать это как дифференциальное уравнение второго порядка, описывающее положение частицы, мы можем ввести переменную $v(t)$, представляющую скорость частицы в момент t, и выразить ньютоновскую динамику в виде уравнения первого порядка:

$$v(t) = \frac{\partial}{\partial t}\theta(t)$$

$$f(t) = \frac{\partial}{\partial t} v(t)$$

Тогда для применения импульсного алгоритма нужно численно решить эту систему дифференциальных уравнений. Простой способ решения дает метод Эйлера, который заключается в моделировании динамики, описываемой уравнением, путем небольших конечных шагов в направлении каждого градиента.

Таким образом, обозначена базовая форма обновления параметров импульсным методом, но остается вопрос: что конкретно представляют собой силы? Одна сила пропорциональна отрицательному градиенту функции стоимости: $-\nabla_\theta J(\theta)$. Эта сила толкает частицу вниз по поверхности функции стоимости. Алгоритм градиентного спуска просто сделал бы один шаг, основанный на градиенте, но в импульсном алгоритме эта сила изменяет скорость частицы. Можно считать частицу хоккейной шайбой, скользящей по ледяной поверхности. Во время спуска по крутому склону она набирает скорость и продолжает скользить в одном направлении, пока не начнется очередной подъем.

Но необходима еще одна сила. Если бы единственной силой был градиент функции стоимости, то частица могла бы никогда не остановиться. Представьте себе шайбу, скользящую вниз по одному склону оврага, затем вверх по противоположному склону – если предположить, что трения нет, то она так и будет опускаться и подниматься бесконечно. Для решения этой проблемы добавлена еще одна сила, пропорциональную $-v(t)$. В физике мы назвали бы ее вязким сопротивлением, как если бы частица должна была прокладывать себе путь в сопротивляющейся среде, например в сиропе. В результате частица постепенно теряет энергию и в конце концов остановится в локальном минимуме.

16. ЛЕКЦИЯ. Алгоритмы с адаптивной скоростью обучения

Специалисты по нейронным сетям давно поняли, что скорость обучения – один из самых трудных для установки гиперпараметров, поскольку она существенно влияет на качество модели. Стоимость зачастую очень чувствительна в некоторых направлениях пространства параметров и нечувствительна в других. Импульсный алгоритм может в какой-то мере сгладить эти проблемы, но ценой введения другого гиперпараметра. Естественно возникает вопрос, нет ли какого-то иного способа. Если мы полагаем, что направления чувствительности почти параллельны осям, то, возможно, имеет смысл задавать скорость обучения отдельно для каждого параметра и автоматически адаптировать эти скорости на протяжении всего обучения.

Алгоритм delta-bar-delta (дельта-бар-дельта) – один из первых эвристических подходов к адаптации индивидуальных скоростей обучения параметров модели. Он основан на простой идее: если частная производная функции потерь по данному параметру модели не меняет знак, то скорость обучения следует увеличить. Если же знак меняется, то скорость следует уменьшить. Конечно, такого рода правило применимо только к оптимизации на полном пакете. Позже был предложен целый ряд инкрементных (увеличивающих и основанных на мини-пакетах) методов для адаптации скоростей обучения параметров.

AdaGrad

Алгоритм AdaGrad (ейдаград) по отдельности адаптирует скорости обучения всех параметров модели, умножая их на коэффициент, обратно пропорциональный квадратному корню из суммы всех прошлых значений квадрата градиента. Для параметров, по которым частная производная функции потерь наибольшая, скорость обучения уменьшается быстро, а если частная производная мала, то и скорость обучения уменьшается медленнее. В итоге больший прогресс получается в направлениях пространства параметров со сравнительно пологими склонами. В случае выпуклой оптимизации у алгоритма AdaGrad есть некоторые желательные теоретические свойства. Но эмпирически при обучении глубоких нейронных сетей накапливание квадратов градиента с самого начала обучения может привести к преждевременному и чрезмерному уменьшению эффективной скорости обучения. AdaGrad хорошо работает для некоторых, но не для всех моделей глубокого обучения.

Алгоритм 3. Алгоритм AdaGrad

Require: глобальная скорость обучения ε

Require: начальные значения параметров θ

Require: небольшая константа δ, например 10^{-7}, для обеспечения численной устойчивости

Инициализировать переменную для агрегирования градиента $r = 0$

while критерий остановки не выполнен **do**

Выбрать из обучающего набора мини-пакет m примеров $\{x^{(1)}, \dots, x(m)\}$ и соответствующие им метки $y^{(i)}$.

Вычислить градиент: $g \leftarrow (1/m)\nabla_\theta \Sigma_i L(f(x^{(i)}; \theta), y^{(i)})$.

Агрегировать квадраты градиента: $r \leftarrow r + g \odot g$.

Вычислить обновление: $\Delta\theta \leftarrow -\frac{\varepsilon}{\delta+\sqrt{r}} \odot g$ (операции деления и извлечения корня применяются к каждому элементу).

Применить обновление: $\theta \leftarrow \theta + \Delta\theta$.

end while

$\odot$ - N-арный оператор. Операция n-арная на множестве g — отображение $f: g^n \to g$, где g^n — декартова n-я степень множества g.

RMSProp

Алгоритм RMSProp (армспроп) – это модификация AdaGrad, призванная улучшить его поведение в невыпуклом случае путем изменения способа агрегирования градиента на экспоненциально взвешенное скользящее среднее. AdaGrad разрабатывался для быстрой сходимости в применении к выпуклой функции. Если же он применяется к невыпуклой функции для обучения нейронной сети, то траектория обучения может проходить через много разных структур и в конечном итоге прийти в локально выпуклую впадину. AdaGrad уменьшает скорость обучения, принимая во внимание всю историю квадрата градиента, и может случиться так, что скорость станет слишком малой еще до достижения такой выпуклой структуры. В алгоритме RMSProp используется экспоненциально затухающее среднее, т. е. далекое прошлое отбрасывается, чтобы повысить скорость сходимости после обнаружения выпуклой впадины, как если бы внутри этой впадины алгоритм AdaGrad был инициализирован заново.

Алгоритм 4. содержит описание RMSProp в стандартной форме. По сравнению с AdaGrad вводится новый гиперпараметр ρ, управляющий масштабом длины при вычислении скользящего среднего.

Алгоритм 4. Алгоритм RMSProp

Require: глобальная скорость обучения ε, скорость затухания ρ

Require: начальные значения параметров θ

Require: небольшая константа δ, например 10^{-6}, для стабилизации

деления на малые числа

Инициализировать переменную для агрегирования градиента $r = 0$

while критерий остановки не выполнен **do**

Выбрать из обучающего набора мини-пакет m примеров $\{x^{(1)}, \dots, x(m)\}$ и соответствующие им метки $y^{(i)}$.

Вычислить градиент: $g \leftarrow (1/m)\nabla_\theta \Sigma_i L(f(x^{(i)}; \theta), y^{(i)})$.

Агрегировать квадраты градиента: $r \leftarrow \rho r + (1-\rho) g \odot g$.

Вычислить обновление: $\Delta\theta \leftarrow -\frac{\varepsilon}{\delta+\sqrt{r}} \odot g$ (операции деления и извлечения корня применяются к каждому элементу).

Применить обновление: $\theta \leftarrow \theta + \Delta\theta$.

end while

Эмпирически показано, что RMSProp – эффективный и практичный алгоритм оптимизации глубоких нейронных сетей. В настоящее время он считается одним из лучших методов оптимизации и постоянно используется в практической работе.

Adam

Adam – еще один алгоритм оптимизации с адаптивной скоростью обучения. Название «Adam» – сокращение от «adaptive moments» (адаптивные моменты). Его, наверное, правильнее всего рассматривать как комбинацию RMSProp и импульсного метода с несколькими важными отличиями. Во-первых, в Adam импульс включен непосредственно в виде оценки первого момента (с экспоненциальными весами) градиента. Самый прямой способ добавить импульс в RMSProp – применить его к масштабированным градиентам. У использования импульса в сочетании с масштабированием нет ясного теоретического обоснования. Во-вторых, Adam включает поправку на смещение в оценки как первых моментов (член импульса), так и вторых (нецентрированных) моментов для учета их инициализации в начале координат (см. алгоритм 8.7). RMSProp также включает оценку (нецентрированного) второго момента, однако в нем нет поправочного коэффициента. Таким образом, в отличие от Adam, в RMSProp оценка второго момента может иметь высокое смещение на ранних стадиях обучения. Вообще говоря, Adam считается довольно устойчивым к выбору гиперпараметров, хотя скорость обучения иногда нужно брать отличной от предлагаемой по умолчанию.

Алгоритм 5. Алгоритм Adam

Require: величина шага ε (по умолчанию 0.001).

Require: коэффициенты экспоненциального затухания для оценок

моментов ρ_1 и ρ_2, принадлежащие диапазону $[0, 1)$ (по умолчанию 0.9 и 0.999 соответственно).

Require: небольшая константа δ для обеспечения численной устойчивости (по умолчанию 10^{-8}).

Require: начальные значения параметров θ.

Инициализировать переменные для первого и второго моментов $s = 0$, $r = 0$

Инициализировать шаг по времени $t = 0$

while критерий остановки не выполнен **do**

Выбрать из обучающего набора мини-пакет m примеров $\{x^{(1)}, \dots, x(m)\}$ и соответствующие им метки $y^{(i)}$.

Вычислить градиент: $g \leftarrow (1/m)\nabla_\theta \Sigma_i L(f(x^{(i)}; \theta), y^{(i)})$.

$t \leftarrow t + 1$

Обновить смещенную оценку первого момента: $s \leftarrow \rho_1 s + (1- \rho_1)g$

Обновить смещенную оценку второго момента: $r \leftarrow \rho_2 r + (1-\rho_2)g \odot g$

Скорректировать смещение первого момента: $\hat{s} \leftarrow \frac{s}{1-\rho_1^t}$

Скорректировать смещение второго момента: $\hat{r} \leftarrow \frac{r}{1-\rho_2^t}$

Вычислить обновление: $\Delta\theta \leftarrow -\frac{\hat{s}}{\delta+\sqrt{\hat{r}}}$ (операции применяются к каждому элементу)

Применить обновление: $\theta \leftarrow \theta + \Delta\theta$.

end while

Выбор правильного алгоритма оптимизации

Таким образом, обсудили ряд родственных алгоритмов, каждый из которых пытается решить проблему оптимизации глубоких моделей, адаптируя скорость обучения каждого параметра. Возникает естественный вопрос: какой алгоритм выбрать? К сожалению, в настоящее время единого мнения нет. Результаты показывают, что семейство алгоритмов с адаптивной скоростью обучения (представленное алгоритмами RMSProp и AdaDelta) ведет себя достаточно устойчиво, явный победитель здесь не выявлен. Сейчас наиболее популярны и активно применяются алгоритмы СГС, СГС с учетом импульса, RMSProp, RMSProp с учетом импульса, AdaDelta и Adam. Какой из них использовать, зависит главным образом от знакомства пользователя с алгоритмом.

Приближенные методы второго порядка

Теперь обсудим применение методов второго порядка к обучению

глубоких сетей. Для простоты будем рассматривать только одну целевую функцию: эмпирический риск:

$$J(\theta) = \mathbb{E}_{x,y\sim\hat{p}_{data(x,y)}}[L(f(x;\theta),y)] = \frac{1}{m}\sum_{i=1}^{m} L(f(x^{(i)};\theta),y^{(i)})$$

Впрочем, рассматриваемые методы легко обобщаются на другие целевые функции.

Метод Ньютона

Мы с Вами познакомились с градиентными методами второго порядка. И знаем, что в отличие от методов первого порядка, в этом случае для улучшения оптимизации задействуются вторые производные. Самый известный метод второго порядка – метод Ньютона. Опишем его более подробно с акцентом на применении к обучению нейронных сетей.

Метод Ньютона основан на использовании разложения в ряд Тейлора с точностью до членов второго порядка для аппроксимации $J(\theta)$ в окрестности некоторой точки θ_0, производные более высокого порядка при этом игнорируются.

$$J(\theta) \approx J(\theta_0 0) + (\theta - \theta_0)^T \nabla_\theta J(\theta_0) + 1/2(\theta - \theta_0)^T H(\theta - \theta_0)$$

где H – гессиан J относительно θ, вычисленный в точке θ_0. Пытаясь найти критическую точку этой функции, приходим к правилу Ньютона для обновления параметров:

$$\theta^* = \theta_0 - H^{-1}\nabla_\theta J(\theta_0)$$

Таким образом, для локально квадратичной функции (с положительно определенной матрицей H) умножение градиента на H^{-1} сразу дает точку минимума. Если целевая функция выпуклая, но не квадратичная (имеются члены более высокого порядка), то это обновление можно повторить, получив тем самым алгоритм обучения, основанный на методе Ньютона.

Для неквадратичных поверхностей метод Ньютона можно применять итеративно, при условии что матрица Гессе остается положительно определенной. Отсюда вытекает двухшаговая итеративная процедура. Сначала обновляем или вычисляем обратный гессиан (путем обновления квадратичной аппроксимации). Затем обновляем параметры в соответствии с правилом Ньютона.

Алгоритм 6. Метод Ньютона с целевой функций

$$\boldsymbol{J(\theta) = \frac{1}{m}\sum_{i=1}^{m} L(f(x^{(i)};\theta),y^{(i)})}$$

Require: начальные значения параметров θ_0.

Require: обучающий набор m примеров

while критерий остановки не выполнен **do**

Вычислить градиент: $g \leftarrow (1/m)\nabla_\theta \sum_i L(f(x^{(i)};\theta), y^{(i)})$

Вычислить гессиан: $H \leftarrow (1/m)\nabla_\theta^2 \sum_i L(f(x^{(i)};\theta), y^{(i)})$

Вычислить обратный гессиан: H^{-1}

Вычислить обновление: $\Delta\theta = -H^{-1}g$

Применить обновление: $\theta \leftarrow \theta + \Delta\theta$.

end while

Вспомним, что метод Ньютона применим, только если матрица Гессе положительно определенная. В глубоком обучении поверхность целевой функции обычно невыпуклая и имеет много особенностей типа седловых точек, с которыми метод Ньютона не справляется. Если не все собственные значения гессиана положительны, например вблизи седловой точки, то метод Ньютона может произвести обновление не в том направлении. Такую ситуацию можно предотвратить с помощью регуляризации гессиана. Одна из распространенных стратегий регуляризации – прибавление константы α ко всем диагональным элементам гессиана. Тогда регуляризированное обновление принимает вид:

$$\theta^* = \theta_0 - [H(f(\theta_0)) + \alpha I]^{-1}\nabla_\theta f(\theta_0)$$

Эта стратегия регуляризации применяется в аппроксимациях метода Ньютона, например в алгоритме Левенберга–Марквардта, и работает неплохо, если отрицательные собственные значения гессиана сравнительно близки к нулю. Если же в некоторых направлениях кривизна сильнее, то значение α следует выбирать достаточно большим для компенсации отрицательных собственных значений. Однако по мере увеличения α в гессиане начинает доминировать диагональ αI, и направление, выбранное методом Ньютона, стремится к стандартному градиенту, поделенному на α. При наличии сильной кривизны α должно быть настолько большим, чтобы метод Ньютона делал меньшие шаги, чем градиентный спуск с подходящей скоростью обучения. Помимо проблем, связанных с такими особенностями целевой функции, как седловые точки, применение метода Ньютона к обучению больших нейронных сетей лимитируется требованиями к вычислительным ресурсам. Число элементов гессиана равно квадрату числа параметров, поэтому при k параметрах метод Ньютона требует обращения матрицы размера $k \times k$. Кроме того, поскольку параметры изменяются при каждом обновлении, обратный гессиан нужно вычислять на каждой итерации обучения. Поэтому лишь сети с очень небольшим числом параметров реально обучить методом Ньютона.

Метод сопряженных градиентов

Метод сопряженных градиентов позволяет избежать вычисления обратного гессиана посредством итеративного спуска в сопряженных направлениях. Идея этого подхода вытекает из внимательного изучения слабого места метода наискорейшего спуска, при котором поиск итеративно производится в направлении градиента. На рис. 3 показано, что метод наискорейшего спуска в квадратичной впадине неэффективен, т. к. продвигается зигзагами. Так происходит, потому что направление линейного поиска, определяемое градиентом на очередном шаге, гарантированно ортогонально направлению поиска на предыдущем шаге.

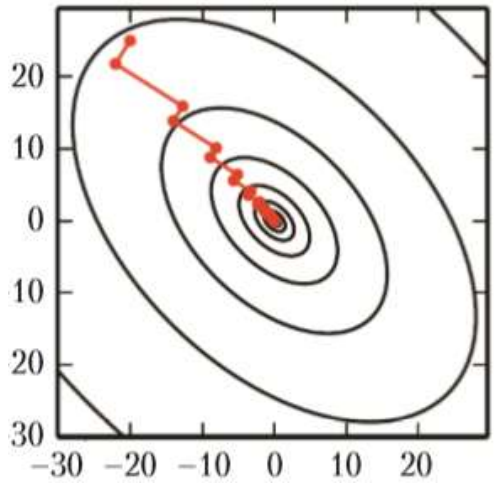

Рис. 3. Метод наискорейшего спуска в применении к поверхности квадратичной целевой функции.

В этом методе на каждом шаге производится переход в точку с наименьшей стоимостью вдоль прямой, определяемой градиентом в начале этого шага. Это решает некоторые показанные на рис. 2 проблемы, которые обусловлены фиксированной скоростью обучения, но даже при оптимальной величине шага алгоритм все равно продвигается к оптимуму зигзагами. По определению, в точке минимума целевой функции вдоль заданного направления градиент в конечной точке ортогонален этому направлению.

Обозначим d_{t-1} направление предыдущего поиска. В точке минимума, где поиск завершается, производная по направлению d_{t-1} равна нулю: $\nabla\theta J(\theta) \times d_{t-1} = 0$. Поскольку градиент в этой точке определяет текущее направление поиска, $dt = \nabla_\theta J(\theta)$ не дает вклада в направлении d_{t-1}. Следовательно, dt ортогонально d_{t-1}. Эта связь между d_{t-1} и dt иллюстрируется на рис. 3 для нескольких итераций метода наискорейшего спуска. Как видно по рисунку, выбор ортогональных направлений спуска не сохраняет минимума вдоль предыдущих направлений поиска. Отсюда и зигзагообразная траектория,

поскольку после спуска к минимуму в направлении текущего градиента должны заново минимизировать целевую функцию в направлении предыдущего градиента. А значит, следуя направлению градиента в конце каждого отрезка, в некотором смысле перечеркивается достигнутое в направлении предыдущего отрезка. Метод сопряженных градиентов и призван решить эту проблему.

В методе сопряженных градиентов направление следующего поиска является сопряженным к направлению предыдущего, т. е. не отказывается от того, что было достигнуто в предыдущем направлении. На t-ой итерации обучения направление следующего поиска определяется формулой:

$$dt = \nabla_\theta J(\theta) + \beta_t d_{t-1},$$

где β_t – коэффициент, определяющий, какую часть направления d_{t-1} следует прибавить к текущему направлению поиска.

Два направления dt и d_{t-1} называются сопряженными, если $d_t^T H d_{t-1} = 0$, где H – матрица Гессе. Самый простой способ обеспечить сопряженность – вычислить собственные векторы H для выбора β_t – не отвечает исходной цели разработать метод, который был бы вычислительно проще метода Ньютона при решении больших задач. Можно ли найти сопряженные направления, не прибегая к таким вычислениям? К счастью, да. Существуют два популярных метода вычисления β_t.

1. Метод Флетчера-Ривса:

$$\beta_t = \frac{\nabla_\theta J(\theta_t)^T \nabla_\theta J(\theta_t)}{\nabla_\theta J(\theta_{t-1})^T \nabla_\theta J(\theta_{t-1})}$$

2. Метод Полака-Рибьера:

$$\beta_t = \frac{(\nabla_\theta J(\theta_t) - \nabla_\theta J(\theta_{t-1}))^T \nabla_\theta J(\theta_t)}{\nabla_\theta J(\theta_{t-1})^T \nabla_\theta J(\theta_{t-1})}$$

Для квадратичной поверхности сопряженность направлений гарантирует, что модуль градиента вдоль предыдущего направления не увеличится. Поэтому минимум, найденный вдоль предыдущих направлений, сохраняется. Следовательно, в k-мерном пространстве параметров метод сопряженных градиентов требует не более k поисков для нахождения минимума.

Нелинейный метод сопряженных градиентов. Мы обсуждали метод сопряженных градиентов в применении к квадратичной целевой функции. Но в нас интересуют в основном методы оптимизации для обучения нейронных сетей и других глубоких моделей, в которых целевая функция далека от квадратичной. Как ни странно, метод сопряженных градиентов применим и в такой ситуации, хотя и с некоторыми изменениями. Если целевая функция не квадратичная, то уже нельзя гарантировать, что поиск в сопряженном направлении сохраняет

минимум в предыдущих направлениях. Поэтому в нелинейном алгоритме сопряженных градиентов время от времени производится сброс, когда метод сопряженных градиентов заново начинает поиск вдоль направления неизмененного градиента.

Алгоритм 7. Метод сопряженных градиентов

Require: начальные значения параметров θ_0.

Require: обучающий набор m примеров

Инициализировать $\rho_0 = 0$

Инициализировать $g_0 = 0$

Инициализировать $t = 1$

while критерий остановки не выполнен **do**

Инициализировать градиент $g_t = 0$

Вычислить градиент: $g_t \leftarrow (1/m)\nabla_\theta \sum_i L(f(x^{(i)};\ \theta), y^{(i)})$

Вычислить $\beta_t = ((g_t - g_{t-1})^T g_t)/g_{t-1}^T g_{t-1}$ (метод Полака–Рибьера)

Вычислить направление поиска: $\rho_t = -g_t + \beta_t \rho_{t-1}$

Произвести поиск с целью нахождения: $\varepsilon^* = argmin_\varepsilon (1/m) \sum_{i=1}^m L(f(x^{(i)};\ \theta_t + \varepsilon\rho_t), y^{(i)})$

Применить обновление: $\theta_{t+1} = \theta_t + \varepsilon^* \rho_t$

$t \leftarrow t + 1$

end while

Есть сообщения, что нелинейный метод сопряженных градиентов дает неплохие результаты при обучении нейронных сетей, хотя часто имеет смысл инициализировать оптимизацию, выполнив несколько итераций стохастического градиентного спуска, и только потом переходить к нелинейным сопряженным градиентам. Кроме того, хотя нелинейный алгоритм сопряженных градиентов традиционно считался пакетным методом, его мини-пакетные варианты успешно применялись для обучения нейронных сетей. Позже были предложены адаптации метода сопряженных градиентов, например алгоритм масштабированных сопряженных градиентов.

Алгоритм BFGS

Алгоритм Бройдена–Флетчера–Гольдфарба–Шанно (BFGS) – попытка взять некоторые преимущества метода Ньютона без обременительных вычислений. В этом смысле BFGS аналогичен методу сопряженных градиентов. Однако в BFGS подход к аппроксимации обновления Ньютона более прямолинейный. Напомним, что обновление Ньютона определяется формулой

$$\theta^* = \theta_0 - H^{-1}\nabla_\theta J(\theta_0)$$

где H – гессиан J относительно θ, вычисленный в точке θ_0. Основная

вычислительная трудность при применении обновления Ньютона – вычисление обратного гессиана H^{-1}. В квазиньютоновских методах (из которых алгоритм BFGS самый известный) обратный гессиан аппроксимируется матрицей M_t, которая итеративно уточняется в ходе обновлений низкого ранга.

После нахождения аппроксимации гессиана M_t направление спуска ρ_t определяется по формуле $\rho_t = M_t g_t$. В этом направлении производится линейный поиск для определения величины шага ε^*. Окончательное обновление параметров производится по формуле

$$\theta_{t+1} = \theta_t + \varepsilon^* \rho_t$$

Как и в методе сопряженных градиентов, в алгоритме BFGS производится последовательность линейных поисков в направлениях, вычисляемых с учетом информации второго порядка. Но, в отличие от метода сопряженных градиентов, успех не так сильно зависит от того, находит ли линейный поиск точку, очень близкую к истинному минимуму вдоль данного направления. Поэтому BFGS тратит меньше времени на уточнение результатов каждого линейного поиска. С другой стороны, BFGS должен хранить обратный гессиан M, для чего требуется память объема $O(n^2)$, поэтому BFGS непригоден для современных моделей глубокого обучения, насчитывающих миллионы параметров.

BFGS в ограниченной памяти (L-BFGS). Потребление памяти в алгоритме BFGS можно значительно уменьшить, если не хранить полную аппроксимацию обратного гессиана M. В алгоритме L-BFGS аппроксимация M вычисляется так же, как в BFGS, но, вместо того чтобы сохранять аппроксимацию между итерациями, делается предположение, что $M^{(t-1)}$ – единичная матрица. При использовании совместно с точным линейным поиском направления, вычисляемые алгоритмом L-BFGS, являются взаимно сопряженными. Однако, в отличие от метода сопряженных градиентов, эта процедура ведет себя хорошо, даже когда линейный поиск находит только приближенный минимум. Описанную стратегию L-BFGS без запоминания можно обобщить, включив больше информации о гессиане; для этого нужно хранить некоторые векторы, используемые для обновления M на каждом шаге, тогда потребуется только память объемом $O(n)$.

Стратегии оптимизации и метаалгоритмы

Многие методы оптимизации – не совсем алгоритмы, а скорее общие шаблоны, которые можно специализировать и получить алгоритмы или подпрограммы, включаемые в различные алгоритмы.

Пакетная нормировка

Пакетная нормировка – одна из наиболее интересных новаций в области

оптимизации глубоких нейронных сетей – вообще алгоритмом не является. Это метод адаптивной перепараметризации (функция называется заменой параметра), появившийся из-за трудностей обучения очень глубоких моделей.

Пакетная нормализация (англ. batch-normalization) — метод, который позволяет повысить производительность и стабилизировать работу искусственных нейронных сетей. Суть данного метода заключается в том, что некоторым слоям нейронной сети на вход подаются данные, предварительно обработанные и имеющие нулевое математическое ожидание и единичную дисперсию.

Нормализация входного слоя нейронной сети обычно выполняется путем масштабирования данных, подаваемых в функции активации. Например, когда есть признаки со значениями от 0 до 1 и некоторые признаки со значениями от 1 до 1000, то их необходимо нормализовать, чтобы ускорить обучение. Нормализацию данных можно выполнить и в скрытых слоях нейронных сетей, что и делает метод пакетной нормализации.

Пакетная нормализация уменьшает величину, на которую смещаются значения узлов в скрытых слоях (т.н. ковариантный сдвиг). Ковариантный сдвиг — это ситуация, когда распределения значений признаков в обучающей и тестовой выборке имеют разные параметры (математическое ожидание, дисперсия и т.д.). Ковариантность в данном случае относится к значениям признаков.

Проиллюстрируем ковариантный сдвиг примером. Пусть есть глубокая нейронная сеть, которая обучена определять находится ли на изображении роза. И нейронная сеть была обучена на изображениях только красных роз. Теперь, если попытаться использовать обученную модель для обнаружения роз различных цветов, то, очевидно, точность работы модели будет неудовлетворительной. Это происходит из-за того, что обучающая и тестовая выборки содержат изображения красных роз и роз различных цветов в разных пропорциях. Другими словами, если модель обучена отображению из множества X в множество Y и если пропорция элементов в X изменяется, то появляется необходимость обучить модель заново, чтобы «выровнять» пропорции элементов в X и Y. Когда пакеты содержат изображения разных классов, распределенные в одинаковой пропорции на всем множестве, то ковариантный сдвиг незначителен. Однако, когда пакеты выбираются только из одного или двух подмножеств (в данном случае, красные розы и розы различных цветов), то ковариантный сдвиг возрастает. Это довольно сильно замедляет процесс обучения модели.

Простой способ решить проблему ковариантного сдвига для входного слоя — это случайным образом перемешать данные перед созданием пакетов. Но для скрытых слоев нейронной сети такой метод не подходит, так как распределение входных данных для каждого узла скрытых слоев изменяется каждый раз, когда происходит обновление параметров в предыдущем слое. Эта проблема называется внутренним ковариантным сдвигом. Для решения данной проблемы часто приходится использовать низкий темп обучения и методы регуляризации при обучении модели. Другим способом устранения внутреннего ковариантного сдвига является метод пакетной нормализации.

Описание метода

Опишем устройство метода пакетной нормализации. Пусть на вход некоторому слою нейронной сети поступает вектор размерности d: $x = (x^{(1)}, \dots, x^{(d)})$. Нормализуем данный вектор по каждой размерности k:

$$\hat{x}^{(1)} = \frac{x^{(k)} - E\left(x^{(k)}\right)}{\sqrt{D(x^{(k)})}}$$

где математическое ожидание и дисперсия считаются по всей обучающей выборке. Такая нормализация входа слоя нейронной сети может изменить представление данных в слое. Чтобы избежать данной проблемы, вводятся два параметра сжатия и сдвига нормализованной величины для каждого $x^{(k)}$: $\gamma^{(k)}$, $\beta^{(k)}$ — которые действуют следующим образом:

$$y^{(k)} = \gamma^{(k)}\hat{x}^{(k)} + \beta^{(k)}$$

Данные параметры настраиваются в процессе обучения вместе с остальными параметрами модели. Пусть обучение модели производится с помощью пакетов B размера m: $B = \{x_1, \dots, x_m\}$. Здесь нормализация применяется к каждому элементу входа с номером k отдельно, поэтому в $x^{(k)}$ индекс опускается для ясности изложения. Пусть были получены нормализованные значения пакета $\hat{x}_1, \dots, \hat{x}_m$. После применения операций сжатия и сдвига были получены $y_1, \dots, y_m$. Обозначим данную функцию пакетной нормализации следующим образом:

$$BN_{\gamma,\beta}: \{x_1, \dots, x_m\} \to \{y1, \dots, ym\}$$

Тогда алгоритм пакетной нормализации можно представить так:

Алгоритм 8. Пакетной нормализации

Вход: значения x из пакета $B = \{x_1, \dots, x_m\}$; настраиваемые параметры γ, β; константа ϵ для вычислительной устойчивости.

Выход: $\{y_i = BN_{\gamma,\beta}(x_i)\}$

$\mu_B = \frac{1}{m}\sum_i^m x_i$ ***// математическое ожидание пакета***

$\sigma_B^2 = \frac{1}{m}\sum_i^m (x_i - \mu_B)^2$ // ***дисперсия пакета***

$\hat{x}_i = \frac{x_i - \mu_B}{\sqrt{\sigma_B^2 + \epsilon}}$ // ***нормализация***

$y_i = \gamma \hat{x}_i + \beta \equiv BN_{\gamma,\beta}(x_i)$ // ***сжатие и сдвиг***

Заметим, что если $\beta = \mu_B$ и $\gamma = \sqrt{\sigma_B^2 + \epsilon}$, то y_i равен x_i, то есть $BN_{\gamma,\beta}(\cdot)$ является тождественным отображением. Таким образом, использование пакетной нормализации не может привести к снижению точности, поскольку оптимизатор просто может использовать нормализацию как тождественное отображение.

Ипользование пакетной нормализации обладает еще несколькими дополнительными полезными свойствами:

- достигается более быстрая сходимость моделей, несмотря на выполнение дополнительных вычислений;
- пакетная нормализация позволяет каждому слою сети обучаться более независимо от других слоев;
- становится возможным использование более высокого темпа обучения, так как пакетная нормализация гарантирует, что выходы узлов нейронной сети не будут иметь слишком больших или малых значений;
- пакетная нормализация в каком-то смысле также является механизмом регуляризации: данный метод привносит в выходы узлов скрытых слоев некоторый шум, аналогично методу dropout (дропаут (англ. dropout) — метод регуляризации нейронной сети для предотвращения переобучения);
- модели становятся менее чувствительны к начальной инициализации весов.

Литература

1. Ясницкий Л.Н. Интеллектуальные системы / Л. Н. Ясницкий – М.: Лаборатория знаний, 2016. – 221 с.

2. Хайкин С. Нейронные сети: полный курс / С. Хайкин; 2-е издание.: Пер. с англ. – М.: Издательский дом «Вильямс», 2006. – 1104 с.

3. Каллан Р. Основные концепции нейронных сетей/ Р. Каллан; Пер. с англ. – М.: Издательский дом «Вильямс», 2001. – 290 с.

4. Бураков, М. В. Нейронные сети и нейроконтроллеры: учеб. пособие / М. В. Бураков. – СПб.: ГУАП, 2013. – 284 с.

5. Круг П. Г. Нейронные сети и нейрокомпьютеры: учеб. пособие / П. Г. Круг – М.: Издательство МЭИ, 2002. – 176 с.

6. Барский А. Б. Нейронные сети: распознавание, управление, принятие решений / А. Б. Барский – М.: Финансы и статистика, 2004. – 176 с.

7.Оссовский С. Нейронные сети для обработки информации / Пер. с пол. И.Д. Рудинского. – М.: Финансы и статистика, 2002. – 344 с.

8. Уоссерман Ф. Нейрокомпьютерная техника: теория и практика / Пер. с англ. Ю.А. Зуев. – М.: Мир, 1992.

9. 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.

10. Джонс М. Т. Программирование искусственного интеллекта в приложениях / М. Т. Джонс; Пер. с англ. Осипов А. И. – М.: ДМК Пресс, 2011. – 312 с.

11. Unity 5.x. Программирование искусственного интеллекта в играх: пер. с англ. Р. Н. Рагимова. - М.: ДМК Пресс, 2017. – 272 с.

12. Рашка С. Python и машинное обучение / С. Рашка; Пер. с англ. А. В. Логунова. – М.: ДМК Пресс, 2017. – 418 с.

13. Сорокин А.Б., Платонова О.Е. Искусственные нейронные сети прямого распространения М.: МИРЭА, 2018. – 64 с. (ISBN 978-5-7339-1470-1)

14. Сорокин А.Б., Железняк Л.М., Зикеева Е.А. Сверточные нейронные сети: примеры реализаций М.: МИРЭА, 2020. – 158 с. (эл. издание)