ЛЕКЦИЯ.

Архитектуры сверточных сетей

Одной из самых популярных глубоких сверточных архитектур (представленная в 2014 году) является модель, которую принято называть VGG. Название происходит от того, что эта модель была разработана в Оксфордском университете в группе визуальной геометрии (Visual Geometry Group). VGG – это сразу две конфигурации сверточных сетей, на 16 и 19 слоев. Основным нововведением стала идея использовать фильтры размером 3×3 с единичным шагом свертки вместо использовавшихся в лучших моделях предыдущих лет сверток с фильтрами 7×7 с шагом 2 и 11×11 с шагом 4 [1]. Причем это хорошо аргументированное предложение:

Во-первых, рецептивное поле трех подряд идущих сверточных слоев размером 3×3 имеет размер 7×7, в то время как весов у них будет всего 27, против 49 в фильтре 7×7. Аналогично обстоит дело с фильтрами 11×11. Это значит, что VGG может стать более глубокой, то есть содержать больше слоев, при этом одновременно уменьшая общее число весов. Конечно между соответствующими сверточными слоями недолжно быть слоев субдискретизации. Во-вторых, наличие дополнительной нелинейности между слоями позволяет увеличить «разрешающую способность» по сравнению с единственным слоем с большей сверткой. Этот же аргумент можно использовать как мотивацию для того, чтобы ввести в сеть свертки размером 1×1; такие слои тоже позволяют добавить дополнительную нелинейность в сеть, не меняя размер рецептивного поля. Схема одной из VGG-сетей показана на рис.23 [2].

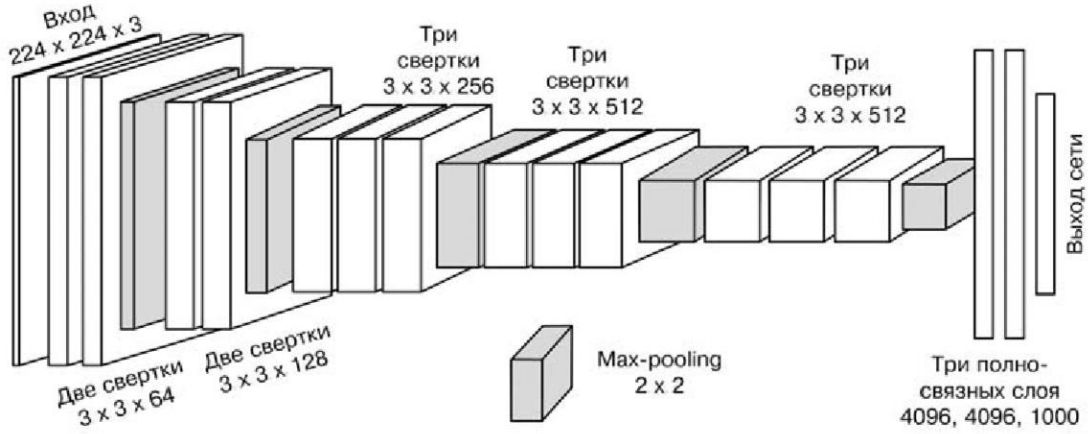

Рис. 23.Схема сети VGG-16

На рис. 23 воплощены три общих принципа [3]:

во-первых, как по две-три свертки 3×3 следуют друг за другом без субдискретизации;

во-вторых, как число карт признаков постепенно растет наболее глубоких уровнях сети;

в-третьих, как в конце полученные признаки окончательно «сплющиваются» в одномерный вектор и на нем работают последние, уже полносвязные слои. Все это стандартные методы создания архитектур глубоких сверточных сетей, и если разрабатывать свою архитектуру, стоит следовать этим общим принципам.

Следующая важная сверточная архитектура – это архитектура Inception (начало) [1]. Она была разработана в Google и появилась практически одновременно с VGG. Авторы вдохновились идеями использовать в качестве строительных блоков для глубоких сверточных сетей непросто последовательность «свертка—нелинейность—субдискретизация», как это обычно делается, а более сложными конструкциями.

В Inception такой компонент «собирают» из небольших сверточных конструкций. Так что в сети развивается идея «вложенной» архитектуры. «Строительными блоками» Inception являются модули, комбинирующие свертки размером 1×1, 3×3 и 5×5, а также max-pooling субдискретизацию. Каждый блок представляет собой объединение четырех «маленьких» сетей, выходы которых объединяются в выходные каналы и передаются в наследующий слой. Выбор набора сверток и субдискретизации обусловлен скорее удобством, чем необходимостью, и при желании можно использовать другие конфигурации. Использование сверточных слоев 1×1 не столько в качестве дополнительной нелинейности, сколько для понижения размерности между слоями. Свертки 3×3 и тем более 5×5 между слоями с большим числом каналов (в Inception-модулях каналов может быть вплоть до 1024), оказываются крайне ресурсоемкими, несмотря на малые размеры отдельно взятых фильтров. Фильтры 1×1 могут помочь сократить число каналов, прежде чем подавать их на фильтры большего размера. Эта идея отражена на рис. 24, а, на котором показана структура одного блока из исходной работы [1]. На рис. 24, б представлена очень общая и высокоуровневая схема всей сети: она начинается с двух «обычных» сверточных слоев, а затем идут 11 Inception-модулей, дважды перемежаемых субдискретизацией, которая понижает размерность; после этого сеть завершается традиционными полносвязными слоями, дающими уже собственно выход классификатора.

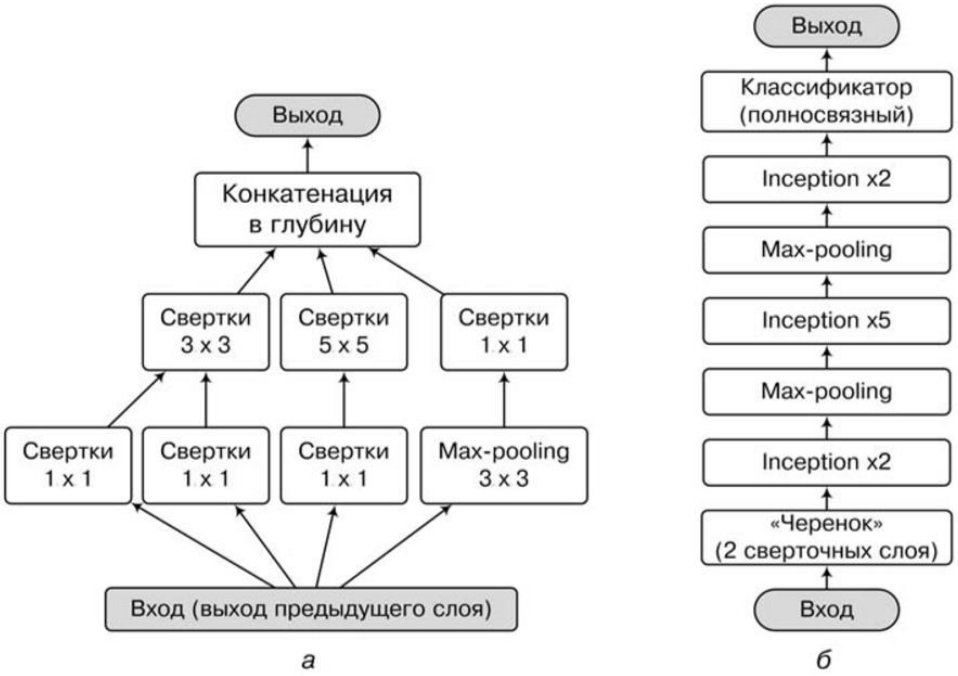

Рис. 24. Inception: а – схема одного Inception-модуля;
б –общая схема сети GoogLeNet

С учетом достаточно глубокой архитектуры сети GoogLeNet – в общей сложности она содержит порядка 100 различных слоев с общей глубиной в 22 параметризованных слоя, или 27 слоев с учетом субдискретизаций – эффективное распространение градиентов по ней вызывает сомнения. Чтобы решить эту проблему, авторы предложили добавить вспомогательные классифицирующие сети поверх некоторых промежуточных слоев. Иначе говоря, добавляются две новые небольшие полносвязные сети, делающие предсказания на основе промежуточных признаков.

Как показала практика, стало возможно глубокие архитектуры обучать эффективно, однако те решения, к которым сходились нейронные сети большой глубины, часто оказывались хуже, чем у менее глубоких моделей. И эта «деградация» не была связана с переобучением, как можно было бы предположить. Оказалось, что это более фундаментальная проблема: с добавлением новых слоев ошибка растет не только на тестовом, но и на тренировочном множестве.

Для решения проблемы деградации команда из Microsoft Research разработала новую идею: глубокое остаточное обучение (deep residual learning), которое легло в основу сети ResNet [1]. В базовой структуре новой модели нет ничего нового: это слои, идущие последовательно друг за другом. Отдельные уровни, составные блоки сети тоже выглядят достаточно стандартно, это просто сверточные слои, обычно с дополнительной нормализацией. Разница в том, что

в остаточном блоке слой из нейронов можно «обойти»: есть специальная связь между выходом предыдущего слоя $x^{(k)}$ и следующего слоя $x^{(k+1)}$, которая идет напрямую, не через вычисляющий слой. Базовый слой нейронной сети на рис. 25, а превращается в остаточный блок с обходным путем на рис. 25, б. Математически происходит очень простая вещь: когда два пути, «сложный» и «обходной», сливаются обратно, их результаты просто складываются друг с другом. И остаточный блок выражает такую функцию:

$$y^{(k)} = F(x^{(k)}) + x^{(k)},$$

где $x^{(k)}$ – входной вектор слоя k, F(x) – функция, которую вычисляет слой нейронов, а $y^{(k)}$ – выход остаточного блока, который потом станет входом следующего слоя $x^{(k+1)}$.

Получается, что если блок в целом должен апроксимировать функцию $H(x)$, то это достигается тогда, когда $F(x)$ аппроксимирует остаток $H(x) - x$, отсюда и название остаточные сети (residual networks). В остаточном блоке обучается слой нейронов воспроизводить изменения входных значений, необходимые для получения итоговой функции.

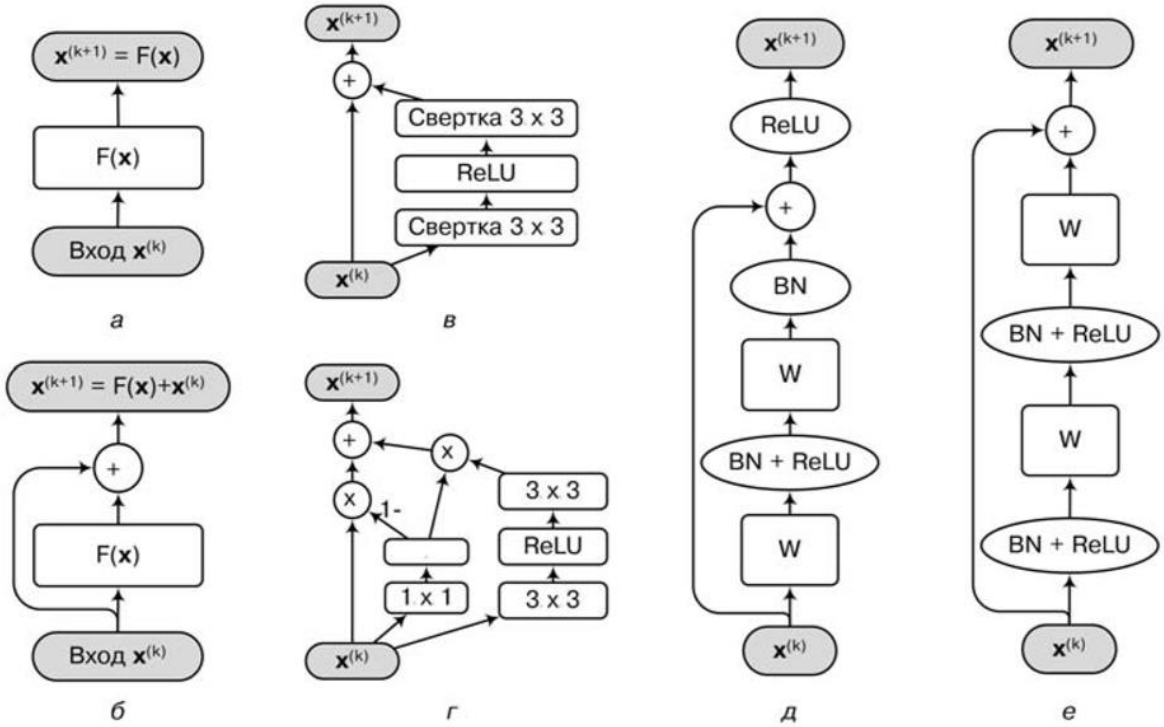

Рис. 25. Блоки сетей для остаточного обучения: а – базовый блок; б – такой же блок с остаточной связью; в – простой блок с остаточной связью; г – блок с остаточной связью, контролирующийся гейтом; д – блок с остаточной связью исходной сети ResNet; е –более поздняя модификация.

Преимущества [1]: Во-первых , часто получается, что обучить «остаточную» функцию проще, чем исходную; Оказывается, что с помощью двухслойной нелинейной нейронной сети выучить тождественную функцию достаточно сложно; в то же время в остаточной форме от сети требуется просто заполнить все веса нулями, а «обходной путь» сделает всю работу сам. Главная же причина состоит в том, что градиент во время обратного распространения может проходить через этот блок беспрепятственно, градиенты не будут затухать, ведь всегда есть возможность пропустить градиент напрямую:

$$\frac{\partial y^{(k)}}{\partial x^{(k)}} = 1 + \frac{\partial F(x^{(k)})}{\partial x^{(k)}}$$

Это значит, что даже насыщенный и полностью обученный слой F, производные которого близки к нулю, не помешает обучению.

Примеры таких «обходных путей», которые использовались в разных вариантах сети ResNet и других исследованиях этой группы авторов, показаны на рис. 25. Проводилось подробное сравнение нескольких вариантов остаточных блоков. Остаточные блоки, которые теоретически должны быть более выразительными, на практике оказываются хуже, чем самые простые варианты.

На рис. 25, в изображен очень простой вариант остаточного блока, в котором $y^{(k)} = F(x^{(k)}) + x^{(k)}$, а на рис. 25, г – вариант посложнее:

$$y^{(k)} = \left(1 - \sigma\left(f(x^{(k)})\right)\right) x^{(k)} + \sigma\left(f(x^{(k)})\right) F(x^{(k)})$$

где f –другая функция входа, реализованная через свертки 1×1.Это значит, что $F(x^{(k)})$ и $x^{(k)}$ суммируются не с равными весами, а с весами, управляемыми дополнительным «гейтом». Казалось бы, простой вариант является частным случаем сложного: достаточно просто обучить гейты так, чтобы веса были равными, то есть чтобы всегда выполнялось $f(x^{(k)}) = 0$. Однако эксперименты в показали, что простота в данном случае важнее выразительности и важно обеспечить максимально свободное и беспрепятственное течение градиентов. На рис. 25, д показан вариант остаточного блока, а на рис. 25, е – улучшенный вариант. Обратите внимание, что разница, по сути, только в том, что из «обходного пути» убрали ReLU-нелинейность, последнее «препятствие» на пути значений с предыдущего слоя [1].

Правда, вне которых приложениях от них отказываются ради скорости и экономии ресурсов: большая сверточная сеть с остаточными связями никак не поместится в смартфон. Если ресурсы важны, стоит посмотреть в сторону моделей, которые показывают немного более слабые результаты в собственно

распознавании, но имеют при этом на порядок меньше весов; выделим, в частности, MobileNets и SqueezeNet.

В заключении отметим, что идея про гейт, управляющий остаточными блоками, как на рис. 25, г, все-таки оказалась довольно плодотворной. Именно она легла в основу так называемых магистральных сетей (highway networks), предложенных группой Юргена Шмидхубера [1]. Идея магистральных сетей именно такая, представляем $y^{(k)}$, выход слоя k, как линейную комбинацию входа этого слоя $x^{(k)}$ и результата $F(x^{(k)})$, веса которой управляются другими преобразованиями [1]:

$$y^{(k)} = C\left(x^{(k)}\right)x^{(k)} + T(x^{(k)})F(x^{(k)})$$

где C – это гейт переноса, а T – гейт преобразования; обычно комбинацию делают выпуклой, $C = 1 - T$. Магистральные сети тоже позволили обучать очень глубокие сети с сотнями уровней, а затем эта конструкция была адаптирована для рекуррентных сетей. Что здесь победит — простота или выразительность — вопрос пока открытый, но в любом случае варианты этих архитектур уже позволили сделать беспрецедентно глубокие сети, и останавливаться на достигнутом исследователи не собираются.

1.8. Библиотеки

В последние годы обучение нейронных сетей превратилось в высшей степени инженерную дисциплину. Появилось несколько очень удобных библиотек, которые позволяют буквально в пару строк кода построить модель нейронной сети (сколь угодно глубокой), сформировать для нее граф вычислений, автоматически подсчитать градиенты и выполнить процесс обучения, причем сделать это можно как на процессоре, так и – совершенно прозрачным образом, изменив пару строк или пару ключей при запуске – на видеокарте, что обычно ускоряет обучение в десятки раз. С каждым годом эти библиотеки становятся все удобнее, выходят новые версии существующих, а также абсолютно новые программные продукты. Есть библиотеки общего назначения, которые могут создать любой граф вычислений, и специализированные надстройки, которые реализуют разные компоненты нейронных сетей.

Библиотеки, которые используются для реализации сверточных нейронных сетей, имеют основной интерфейс к языку программирования Python. Этот язык стал фактически стандартом в современном машинном обучении и обработке данных.

Keras (*https://keras.io*) – это фреймворк поддержки глубокого обучения для Python, обеспечивающий удобный способ создания и обучения практически

любых моделей глубокого обучения. Первоначально Keras разрабатывался для исследователей с целью дать им возможность быстро проводить эксперименты. Keras обладает следующими ключевыми характеристиками [4]:

- позволяет выполнять один и тот же код на CPU или GPU;
- имеет дружественный API, упрощающий разработку прототипов моделей глубокого обучения;
- включает в себя встроенную поддержку сверточных сетей (для распознавания образов), рекуррентных сетей (для обработки последовательностей) и все возможных их комбинаций;
- включает в себя поддержку произвольных сетевых архитектур: моделей с множественными входами или выходами, совместное использование слоев, совместное использование моделей и т. д. Это означает, что Keras подходит для создания практически любых моделей глубокого обучения, от порождающих состязательных сетей до нейронной машины Тьюринга.

Фреймворк Keras распространяется на условиях свободной лицензии и может бесплатно использоваться в коммерческих проектах. Он совместим с любыми версиями Python. Keras используется в Google, Netflix, Uber, CERN, Yelp, Square и сотнях стартапов, решающих широкий круг задач.

Keras – это библиотека уровня модели, предоставляющая высокоуровневые строительные блоки для конструирования моделей глубокого обучения. Она не реализует низкоуровневые операции, такие как манипуляции с тензорами и дифференцирование, – для этого используется специализированная и оптимизированная библиотека поддержки тензоров. При этом Keras не полагается на какую-то одну библиотеку поддержки тензоров, а использует модульный подход (рис. 26); то есть к фреймворку Keras можно подключить несколько разных низкоуровневых библиотек [5].

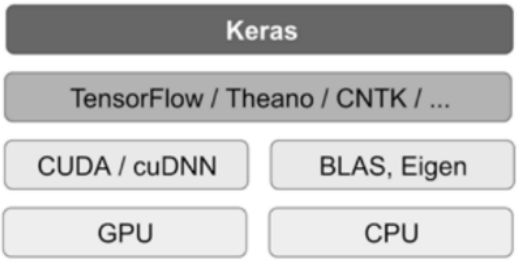

Рис. 26. Программно-аппаратный стек поддержки глубокого обучения

В настоящее время поддерживаются три такие библиотеки: TensorFlow, Theano и Microsoft Cognitive Toolkit (CNTK). В будущем Keras, скорее всего, будет расширен еще несколькими низкоуровневыми механизмами поддержки глубокого обучения.

TensorFlow, CNTK и Theano – это одни из ведущих платформ глубокого обучения в настоящее время. Theano (*http://deeplearning.net/software/theano*) разработана в лаборатории MILA Монреальского университета, TensorFlow (*www.tensorflow.org*) разработана в Google, а CNTK (*https://github.com/Microsoft/CNTK*) разработана в Microsoft. Любой код, использующий Keras, можно запускать с любой из этих библиотек без необходимости менять что-либо в коде: можно легко переключаться между ними в процессе разработки, что часто оказывается полезно, например, если одна из библиотек показывает более высокую производительность при решении данной конкретной задачи. Рекомендуется по умолчанию использовать библиотеку TensorFlow как наиболее распространенную, масштабируемую и высококачественную [5].

Среди библиотек общего назначения, которые способны строить граф вычислений и проводить автоматическое дифференцирование, долгое время бесспорным лидером была Theano.. Однако в ноябре 2015 года Google выпустила (с открытым исходным кодом) библиотеку TensorFlow [3], предназначенную для таких же целей. TensorFlow стала вторым поколением библиотек глубокого обучения в Google. В TensorFlow реализован полный набор операций над тензорами из NumPy с поддержкой матричных вычислений над массивами разной формы и конвертирования между этими формами. Библиотеки Theano и TensorFlow на данный момент остаются двумя бесспорными лидерами в этой области, и сложно уверенно рекомендовать одну из них [5].

Используя TensorFlow (Theano или CNTK), Keras может выполнять вычисления и на CPU, и на GPU. При выполнении на CPU TensorFlow сама использует низкоуровневую библиотеку специализированных операций с тензорами, которая называется Eigen (*http://eigen.tuxfamily.org*). При выполнении на GPU TensorFlow использует оптимизированную библиотеку под названием NVIDIA CUDA DeepNeural Network (cuDNN).

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.

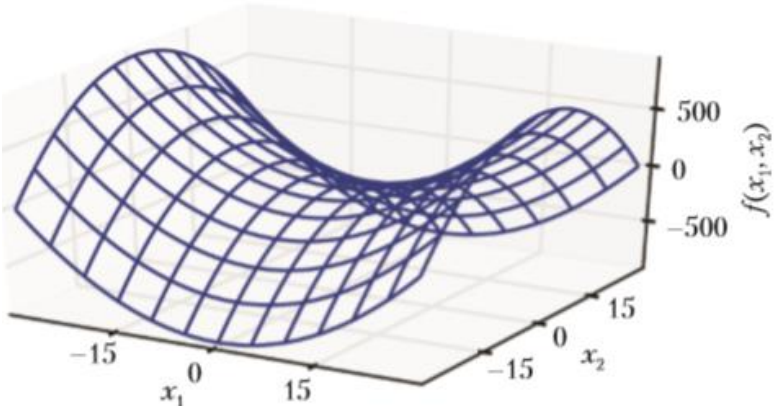

Рис. 1. График функции $f(x) = x_1^2 - x_2^2$

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.

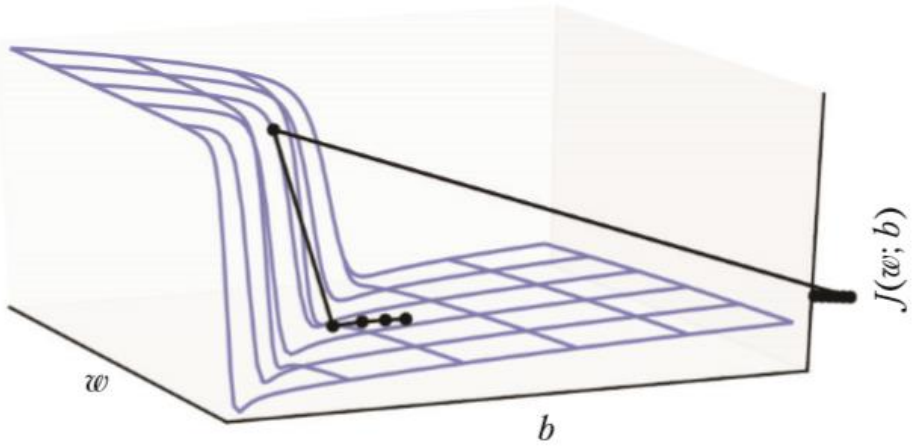

Рис. 2. Утес

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.

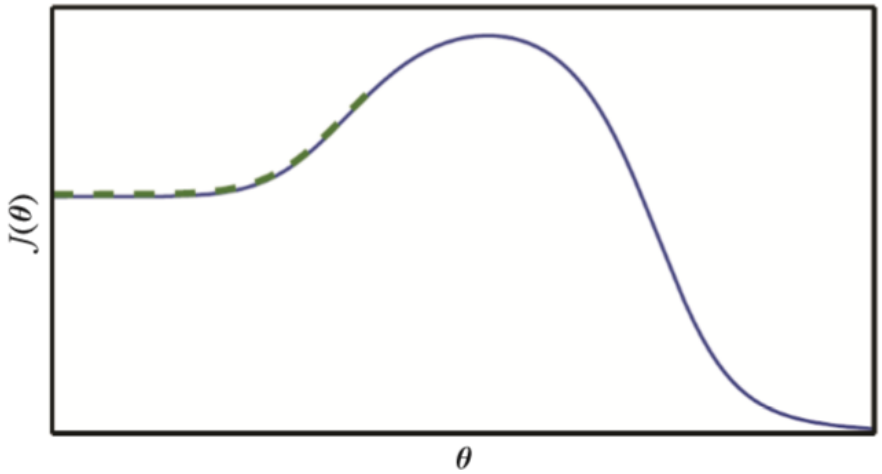

Рис. 3. Траектория обучения

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.

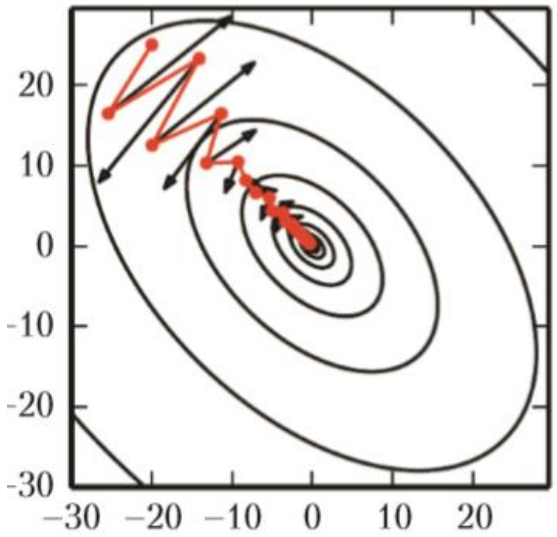

Рис.1. Работа импульсного метода

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.

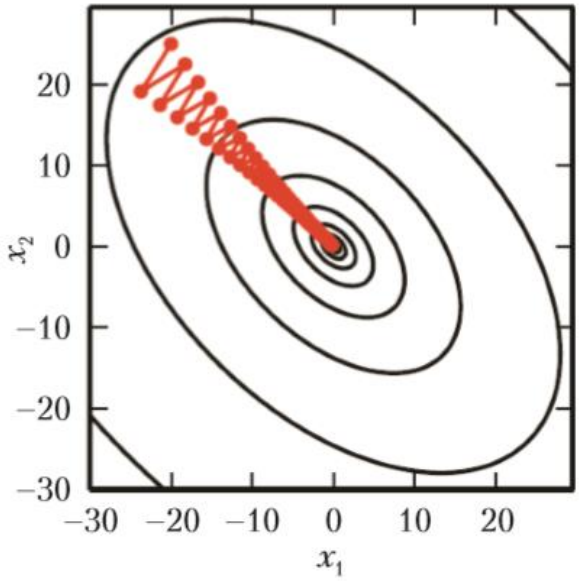

Рис.2. Работа градиентного метода

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.