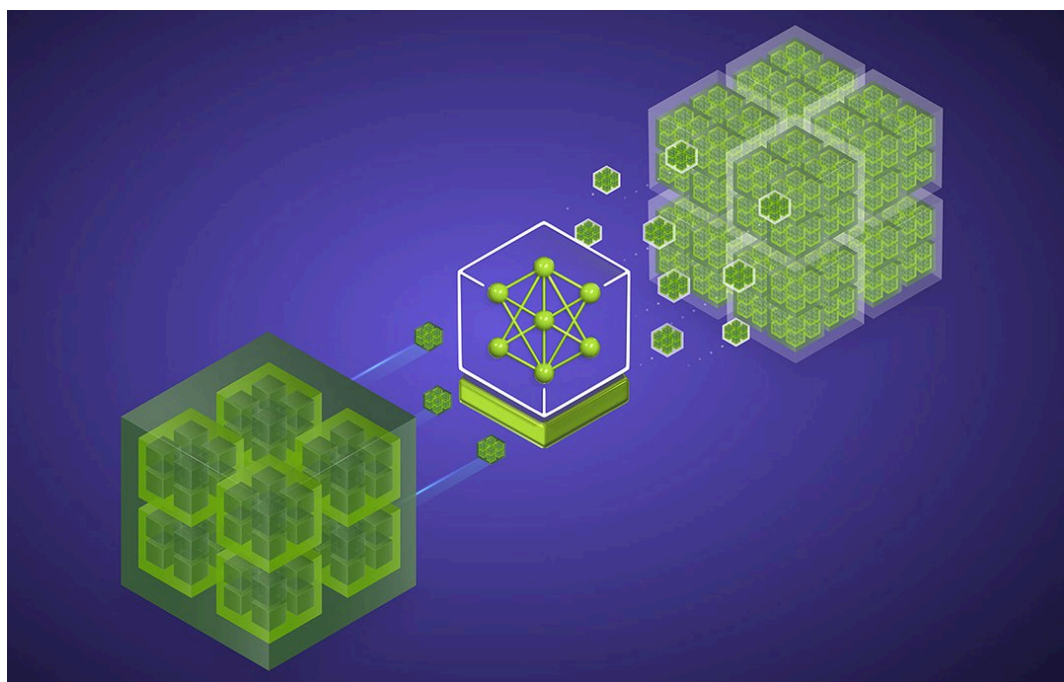


Искусственные нейронные сети

Лекционные материалы



Содержание

Введение	3
Лекция 1. Основные теоремы нейроинформатики	9
Лекция 2. Формальное описание искусственных нейронных сетей .	18
Лекция 3. Вычисления на графе	31
Лекция 4. Задача обучения нейронных сетей	37
Лекция 5. Общая схема градиентного обучения	53
Лекция 6. Автоматическое дифференцирование для обучения НС ..	61
Лекция 7. Практические методы градиентного обучения	73
Лекция 8. Настройка обучения и выбор оптимизатора	87
Лекция 9. Стандартные вычислительные блоки нейросетей	100
Лекция 10. Методы регуляризации	112
Лекция 11. Обучение без учителя	122
Лекция 12. Сверточные сети для обработки изображений	134
Лекция 13. Рекуррентные сети	144
Лекция 14. Трансформеры	152
Лекция 15. Обучение с подкреплением	167
Лекция 16. Диффузионные модели	176
Лекция 17. Байесовы нейронные сети	184
Приложение А. Теорема Колмогорова о представимости функций .	193
Приложение Б. Фундаментальные теоремы нейровычислений	197
Приложение В. Вычисление производных	201

Введение

Искусственный интеллект сегодня нельзя сводить только к экспертным системам или к попытке построить «нейрокомпьютер» как отдельный тип вычислительной машины. Современный ИИ развивается как совокупность методов, которые позволяют строить модели поведения, восприятия, прогнозирования и генерации на основе данных. В этой совокупности центральное место занимает машинное обучение, а внутри него — глубокое обучение: подход, в котором модель автоматически настраивает большое число параметров и строит многоуровневые представления исходных данных [1,2].

Классическое программирование требует заранее описать правила решения задачи. Нейросетевой подход устроен иначе: разработчик задает архитектуру модели, функцию потерь, данные и процедуру оптимизации, а конкретные признаки и параметры находятся в процессе обучения. Поэтому глубокие нейронные сети особенно важны там, где невозможно вручную выписать полный набор правил: при распознавании изображений и речи, машинном переводе, медицинской диагностике, управлении роботами, поиске закономерностей в больших массивах данных и генерации текста, изображений, звука или программного кода.

От нейровычислений к глубокому обучению

Термин «нейровычисления» исторически подчеркивал сходство искусственных нейронных сетей с распределенной обработкой информации в биологических нервных системах. В ранних работах акцент часто делался на специальной аппаратной реализации: нейрочипах, параллельных платах, оптоэлектронных схемах. Этот взгляд был важен, но теперь он стал лишь частью более широкой картины.

Современная парадигма опирается не только на аппаратную параллельность, но и на три взаимосвязанных фактора:

1. Большие наборы данных, в которых модель может найти статистические закономерности.
2. Массово-параллельные вычисления на GPU, TPU и других ускорителях.
3. Алгоритмы обучения, позволяющие эффективно оптимизировать глубокие модели с миллионами и миллиардами параметров.

Глубокое обучение полезно не потому, что оно буквально копирует мозг, а потому что оно дает практический способ обучать сложные параметрические функции. Такие функции преобразуют исходные данные в последовательность все более абстрактных представлений: от пикселей к границам и объектам в компьютерном зрении, от символов и слов к смысловым зависимостям в обработке языка, от сенсорных сигналов к решениям в системах управления [1,2].

Исторические вехи

Развитие нейронных сетей шло волнами: периоды сильных ожиданий сменялись периодами разочарования, пока не появились достаточные данные, вычисления и методы оптимизации.

- **1943** — Уоррен Маккалок и Уолтер Питтс предложили формальную модель искусственного нейрона как логического вычислительного элемента.
- **1949** — Дональд Хебб сформулировал идею обучения через усиление связей между совместно активными нейронами.
- **1958** — Фрэнк Розенблатт представил персептрон, одну из первых обучаемых нейросетевых моделей.
- **1969** — Марвин Минский и Сеймур Пейперт показали ограничения однослойного персептрона; интерес к нейросетям временно снизился.
- **1974** — метод обратного распространения ошибки был впервые описан Александром Галушкиным, а также независимо и одновременно Полом Вербосом; это создало математическую основу для последующего обучения многослойных сетей [12,13].
- **1982** — Джон Хопфилд исследовал рекуррентные сети с коллективной динамикой и энергетической интерпретацией.
- **1986** — Дэвид Румельхарт, Джеффри Хинтон и Рональд Уильямс популяризовали обратное распространение ошибки для обучения многослойных сетей [3].
- **1989-1998** — Ян Лекун и соавторы развили сверточные нейронные сети для распознавания рукописных символов; архитектура LeNet стала ранним примером успешного глубокого обучения в прикладной задаче.
- **2006** — работы Хинтона и коллег вернули интерес к глубоким моделям через предварительное обучение и глубокие вероятностные архитектуры.

- **2012** — AlexNet резко улучшила качество классификации изображений ImageNet за счет глубокой сверточной сети, GPU-обучения, ReLU и регуляризации [4].
- **2014** — появились генеративно-сопоставительные сети, где генератор и дискриминатор обучаются в конкурентной постановке [5]; в машинном переводе внимание позволило модели выбирать релевантные части входной последовательности [6].
- **2015-2016** — ResNet показала, что остаточные связи позволяют обучать очень глубокие сети [7]; AlphaGo объединила глубокие нейросети, обучение с подкреплением и поиск по дереву для игры го [8].
- **2017** — архитектура трансформера (Transformer) заменила рекуррентную обработку механизмом самовнимания (self-attention) и стала основой большинства современных больших языковых моделей [9].
- **2018** — BERT показал эффективность предварительного обучения трансформеров и последующей донастройки для широкого класса задач обработки языка [10].
- **2020** — GPT-3 продемонстрировала, что масштабирование языковых моделей усиливает способности к решению без примеров (zero-shot) и по нескольким примерам (few-shot): модель может решать новые задачи по текстовой инструкции или несколькими примерами без изменения весов [11].

Современные преимущества нейросетевой парадигмы

Главное преимущество глубокого обучения — автоматическое построение признаков. В классическом машинном обучении инженер часто вручную проектирует признаки: края изображения, статистики сигнала, словари терминов, правила обработки текста. Глубокая модель учится извлекать такие признаки сама, причем на разных уровнях абстракции. Это сокращает объем ручной разработки и позволяет работать с сырыми данными: изображениями, аудио, текстами, временными рядами и графами.

Второе преимущество — масштабируемость. Современные нейронные сети часто улучшаются при росте данных, вычислительных ресурсов и числа параметров. Это не означает, что «больше» всегда автоматически «лучше»: важны качество данных, архитектура, оптимизация, регуляризация и проверка на независимых тестах. Однако именно мас-

штабирование стало одной из причин прорыва в компьютерном зрении, распознавании речи, машинном переводе и генеративном ИИ [1,11].

Третье преимущество — универсальность архитектурных блоков. Свертки используются для локальных пространственных структур, рекуррентные модели и механизмы внимания — для последовательностей, трансформеры — для языка, изображений, аудио и мультимодальных данных. Один и тот же общий принцип обучения параметрической модели через функцию потерь применим к разным предметным областям.

Четвертое преимущество — сквозное обучение (end-to-end learning). Вместо конвейера из множества вручную настроенных этапов можно обучать модель, которая напрямую связывает вход и целевой результат. Например, система распознавания речи может переходить от звукового сигнала к тексту, а система машинного перевода — от предложения на одном языке к предложению на другом. Это не отменяет инженерную работу, но меняет ее характер: ключевыми становятся данные, постановка задачи, контроль качества, вычислительная эффективность и безопасность.

Пятое преимущество — появление генеративных и интерактивных моделей. Современные нейросети не только классифицируют или прогнозируют, но и создают новые объекты: тексты, изображения, музыку, программный код, планы действий. Большие языковые модели стали интерфейсом к знаниям и вычислительным инструментам, а мультимодальные модели объединяют текст, изображения, звук и видео. В то же время эти системы могут ошибаться, наследовать смещения обучающих данных, генерировать недостоверные ответы и требовать значительных вычислительных ресурсов. Поэтому современное изучение нейронных сетей обязательно включает не только архитектуры и алгоритмы, но и вопросы надежности, интерпретируемости, этики и контроля качества.

Что такое искусственная нейронная сеть

Искусственная нейронная сеть — это параметрическая вычислительная модель, состоящая из связанных слоев или модулей. Каждый слой выполняет преобразование входных данных, обычно зависящее от обучаемых параметров. Между слоями применяются нелинейные функции активации, благодаря которым сеть может описывать сложные зависимости.

Обучение сети — это задача оптимизации. На вход подаются данные, модель делает предсказание, функция потерь измеряет ошибку, а алгоритм оптимизации изменяет параметры так, чтобы уменьшить эту ошибку. В большинстве современных глубоких сетей для вычисления производных используется обратное распространение ошибки, а для обновления параметров — варианты стохастического градиентного спуска.

Эта дисциплина находится на пересечении нескольких областей:

- математики: линейной алгебры, анализа, теории вероятностей, оптимизации;
- информатики: алгоритмов, архитектур вычислений, систем хранения и обработки данных;
- инженерии данных: подготовки, разметки, очистки и проверки обучающих выборок;
- прикладных областей: медицины, промышленности, финансов, образования, робототехники, анализа текста и изображений.

В последующих лекциях будут рассматриваться математические основы нейронных сетей, функции потерь, методы оптимизации, архитектуры и практические ограничения обучения моделей.

Вопросы для самопроверки

1. Чем современное глубокое обучение отличается от раннего представления о нейровычислениях как о специальной аппаратной парадигме?
2. Почему однослойный персептрон оказался ограниченной моделью и почему многослойные сети смогли преодолеть часть этих ограничений?
3. Какую роль сыграло обратное распространение ошибки в развитии многослойных нейронных сетей?
4. Почему AlexNet считается важной вехой в развитии глубокого обучения?
5. В чем состоит идея механизма внимания и почему трансформер оказался важен для обработки последовательностей?
6. Какие преимущества дает автоматическое извлечение признаков по сравнению с ручным проектированием признаков?
7. Почему рост данных, вычислений и числа параметров улучшает многие современные модели, но не гарантирует качества сам по себе?

8. Какие риски и ограничения нужно учитывать при применении современных генеративных моделей?

Источники

1. Zhang A., Lipton Z. C., Li M., Smola A. J. [Dive into Deep Learning: Introduction](#).
2. Китов В. [Введение в глубокое обучение](#).
3. Rumelhart D. E., Hinton G. E., Williams R. J. [Learning representations by back-propagating errors](#). Nature, 1986.
4. Krizhevsky A., Sutskever I., Hinton G. E. [ImageNet Classification with Deep Convolutional Neural Networks](#). NeurIPS, 2012.
5. Goodfellow I. et al. [Generative Adversarial Nets](#). NeurIPS, 2014.
6. Bahdanau D., Cho K., Bengio Y. [Neural Machine Translation by Jointly Learning to Align and Translate](#). ICLR, 2015.
7. He K., Zhang X., Ren S., Sun J. [Deep Residual Learning for Image Recognition](#). CVPR, 2016.
8. Silver D. et al. [Mastering the game of Go with deep neural networks and tree search](#). Nature, 2016.
9. Vaswani A. et al. [Attention Is All You Need](#). NeurIPS, 2017.
10. Devlin J., Chang M.-W., Lee K., Toutanova K. [BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding](#). NAACL, 2019.
11. Brown T. et al. [Language Models are Few-Shot Learners](#). NeurIPS, 2020.
12. Галушкин А. И. [Синтез многослойных систем распознавания образов](#). Москва: Энергия, 1974.
13. Werbos P. J. [Beyond Regression: New Tools for Prediction and Analysis in the Behavioral Sciences](#). Ph.D. thesis, Harvard University, 1974.

Лекция 1. Основные теоремы нейроинформатики

Лекция вводит теоретические основания нейроинформатики: представление непрерывных функций, универсальную аппроксимацию и связь этих результатов с архитектурами нейронных сетей. Основное внимание уделяется тому, почему нелинейные элементы и суперпозиции позволяют строить богатые классы функций.

1.1. Постановка проблемы

Нейронные сети применяют как универсальные классы параметризованных функций, но это утверждение требует математического обоснования. Нужно понять, какие функции можно выразить точно, какие можно аппроксимировать с заданной точностью и какие свойства операций делают нейросетевые модели достаточно выразительными.

1.2. Зачем нужны теоремы представления и аппроксимации

Нейронные сети используются как классы функций: они принимают входные данные, выполняют линейные операции, применяют нелинейные преобразования и строят сложные суперпозиции. Поэтому один из первых теоретических вопросов нейроинформатики звучит так:

«какие функции в принципе можно представить или приблизить с помощью операций, похожих на вычисления в нейронной сети?»

В этой лекции рассматриваются два типа ответов.

1. **Точное представление** непрерывной функции многих переменных через функции одного переменного. Такой результат связан с теоремой Колмогорова.
2. **Приближенное представление** непрерывной функции с любой заданной точностью. Такой результат связан с теоремами Стоуна, Стоуна-Вейерштрасса и их обобщениями.

Для нейронных сетей особенно важен второй тип результата: на практике вычисления почти всегда приближенные, а задача обычно состоит не в получении единственной точной формулы, а в построении модели с достаточно малой ошибкой.

1.3. Теорема Колмогорова о представлении непрерывных функций

1.3.1. Исторический контекст

В списке проблем, которые Д. Гильберт предложил как ориентиры развития математики XX века, тринадцатая проблема была связана с представимостью функций нескольких переменных через функции меньшего числа переменных.

Один из вариантов вопроса: можно ли корень уравнения

$$x^7 + ax^3 + bx^2 + cx + 1 = 0 \quad 1.1$$

рассматривать как функцию коэффициентов и представить эту функцию суперпозицией непрерывных функций двух переменных?

А. Н. Колмогоров дал ответ на более общую задачу: можно ли произвольную непрерывную функцию нескольких переменных получить с помощью сложения и суперпозиции из непрерывных функций одного переменного. В работах А. Н. Колмогорова и В. И. Арнольда было показано, что такая представимость возможна.

1.3.2. Содержательный смысл результата

Теорема Колмогорова утверждает, что непрерывная функция многих переменных может быть точно записана через конечное число непрерывных функций одного переменного и операций сложения. Это важный результат для нейроинформатики, потому что нейронные сети также строят сложные функции из простых нелинейностей и линейных комбинаций.

1.3.3. Формулировка теоремы Колмогорова

Теорема Колмогорова (О представимости непрерывных функций). При любом целом $n \geq 2$ существуют непрерывные действительные функции

$$\psi_{pq} : [0, 1] \rightarrow \mathbb{R}, \quad 1.2$$

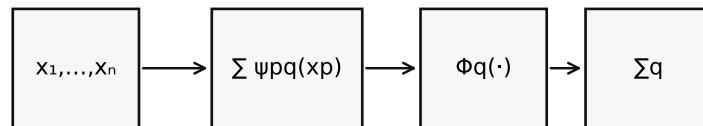
которые не зависят от выбираемой функции f , такие что каждая непрерывная действительная функция

$$f : [0, 1]^n \rightarrow \mathbb{R} \quad 1.3$$

представима в виде

$$f(x_1, \dots, x_n) = \sum_{q=0}^{2n} \Phi_q \left(\sum_{p=1}^n \psi_{pq}(x_p) \right), \quad 1.4$$

где функции $\Phi_q : \mathbb{R} \rightarrow \mathbb{R}$ являются непрерывными и зависят от функции f .



суперпозиция функций одной переменной дает представление многомерной функции

Рис. 1.1. Представление Колмогорова сводит многомерную зависимость к суммам и суперпозициям одномерных функций.

1.3.4. Что важно в формулировке

- Внутренние функции ψ_{pq} универсальны для фиксированного числа переменных n .
- Внешние функции Φ_q подбираются под конкретную функцию f .
- Представление является точным в классе непрерывных функций на единичном кубе.
- Теорема показывает принципиальную возможность сведения многомерной непрерывной зависимости к суперпозициям одномерных непрерывных функций.

1.4. Переход от точного представления к аппроксимации

Точное представление важно теоретически, но для вычислительной практики не менее важен вопрос аппроксимации:

можно ли приблизить произвольную непрерывную функцию многих переменных с любой заданной точностью с помощью функций более простого вида?

Этот вопрос естественен для нейронных сетей. При обучении сети обычно не требуется точная аналитическая формула целевой функции.

Достаточно, чтобы выбранный класс моделей был достаточно богатым и мог приближать нужные зависимости.

1.5. Базовые определения

Компакт. Подмножество M пространства X называется компактным, если из любого открытого покрытия множества M можно выбрать конечное подпокрытие.

Нормальное пространство. Пространство X называется нормальным, если для любых двух непересекающихся замкнутых множеств $A, B \subset X$ существуют непересекающиеся открытые множества $U, V \subset X$ такие, что

$$A \subset U, \quad B \subset V. \quad 1.5$$

Пространство непрерывных функций. Через $C(X)$ обозначают множество всех вещественных непрерывных функций, определенных на X .

Разделение точек. Множество функций $B \subset C(X)$ разделяет точки пространства X , если для любой пары различных точек $x_1, x_2 \in X$ существует функция $g \in B$ такая, что

$$g(x_1) \neq g(x_2). \quad 1.6$$

Плотность. Подмножество $B \subset C(X)$ плотно в $C(X)$, если любую функцию из $C(X)$ можно сколь угодно точно приблизить функциями из B в выбранной норме, обычно в норме равномерного приближения.

1.6. Теорема Стоуна-Вейерштрасса

Классическая теорема Стоуна-Вейерштрасса обобщает идею приближения непрерывных функций многочленами.

Теорема Стоуна-Вейерштрасса. Пусть X - компактное пространство, $C(X)$ - множество всех вещественных непрерывных функций на X , а $B \subset C(X)$ удовлетворяет условиям:

1. если $f, g \in B$, то произведение fg и любые линейные комбинации $\alpha f + \beta g$ с вещественными коэффициентами также принадлежат B ;
2. постоянная функция 1 принадлежит B ;
3. функции из B разделяют точки пространства X .

Тогда B плотно в $C(X)$ в норме равномерного приближения:

$$\overline{B} = C(X). \quad 1.7$$

Иначе говоря, для любой функции $f \in C(X)$ и любого $\varepsilon > 0$ существует функция $g \in B$ такая, что

$$\max_{x \in X} |f(x) - g(x)| < \varepsilon. \quad 1.8$$

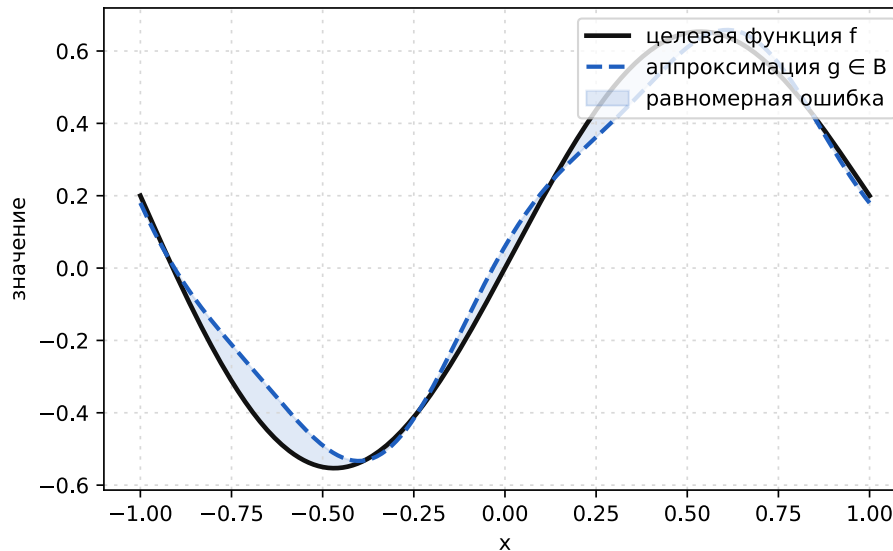


Рис. 1.2. Равномерная аппроксимация требует малой ошибки на всей компактной области.

1.6.1. Смысл теоремы

Теорема говорит, что если класс функций достаточно богат алгебраически, содержит константы и умеет различать точки пространства, то его равномерное замыкание совпадает со всем пространством непрерывных функций. В терминах аппроксимации это означает, что функции из такого класса могут приблизить любую непрерывную зависимость на компакте с любой заданной точностью.

1.7. Обобщенная теорема Стоуна

В контексте нейроинформатики особенно важны классы функций, получаемые с помощью линейных операций и суперпозиций функций одного переменного. Обобщенная теорема Стоуна показывает, что при наличии нелинейной унарной операции такой класс оказывается достаточно богатым для аппроксимации непрерывных функций.

1.7.1. Формулировка

Обобщенная теорема Стоуна. Пусть X - компактное пространство, $C(X)$ - пространство непрерывных вещественных функций на X , $C(\mathbb{R})$ - пространство непрерывных функций на действительной оси, а $H \subset C(X)$ - замкнутое линейное подпространство.

Пусть выполнены условия:

1. постоянная функция 1 принадлежит H ;
2. функции из H разделяют точки пространства X ;
3. существует такая непрерывная нелинейная унарная операция $\varphi \in C(\mathbb{R})$, что замыкание H относительно линейных комбинаций и суперпозиций с φ порождает достаточно богатую алгебру функций, содержащую константы и разделяющую точки.

Тогда

$$\overline{H} = C(X). \quad 1.9$$

В этой лекции это утверждение используется как мост к идее универсальной аппроксимации, а не как полный технический критерий для любой произвольной нелинейности.

1.7.2. Смысл для нейронных сетей

Нейронная сеть строит функции из трех базовых типов операций:

1. линейные комбинации;
2. нелинейные функции активации одного аргумента;
3. суперпозиции слоев и блоков.

Если нелинейность достаточно богата, а построенный класс функций содержит константы и разделяет точки входного пространства, то его замыкание может совпадать с пространством непрерывных функций на компакте. Поэтому нелинейные активации важны не сами по себе, а вместе с линейными комбинациями, суперпозициями и условиями плотности.

1.8. Следствие для нейроинформатики

Следствие. Для многих стандартных семейств нейронных сетей с фиксированной нелинейной активацией и достаточным числом элементов класс реализуемых функций плотен в пространстве непрерывных функций на компакте.

В языке нейронных сетей это означает универсальную аппроксимирующую способность при явно заданных условиях: входная область компактна, точность измеряется равномерной нормой, а функция активации и архитектурный класс удовлетворяют условиям соответствующей теоремы универсальной аппроксимации. Например, для сигмоидальных активаций такой результат формализован в теореме Цыбенко.

1.9. Связь с дальнейшим курсом

Теоремы представления и аппроксимации не дают готового алгоритма обучения, но задают важную границу возможного. Они отвечают на вопрос, есть ли у выбранного класса моделей достаточно выразительности, чтобы в принципе описать нужную зависимость.

Дальнейшие лекции добавляют к этому ответу несколько практических уровней.

1. Формальное описание искусственной нейронной сети уточняет, из каких множеств, функций и матриц связей строится модель.
2. Вычисления на графе показывают, как значения проходят через структуру сети.
3. Обучение как оптимизация объясняет, как подобрать параметры модели по данным.
4. Градиентные методы и автоматическое дифференцирование дают вычислительный механизм поиска параметров.
5. Регуляризация и выбор архитектур помогают избежать ситуации, когда выразительная модель просто запоминает обучающую выборку.

Поэтому универсальная аппроксимация является необходимой, но не достаточной частью теории нейронных сетей. Она говорит, что хорошее приближение существует при подходящих условиях, но не гарантирует, что оптимизационный алгоритм найдет это приближение по конечной выборке.

Из этой разницы следуют три отдельных вопроса курса:

- выразительность архитектуры;
- обучаемость параметров;
- способность обобщать результат за пределы обучающей выборки.

1.10. Итог

- Теорема Колмогорова отвечает на вопрос о точном представлении непрерывных функций многих переменных через функции одного переменного.
- Теорема Стоуна-Вейерштрасса задает общий критерий, при котором равномерное замыкание класса функций совпадает со всем пространством непрерывных функций на компакте.
- Обобщенные результаты Стоуна связывают линейные операции, суперпозиции и нелинейные унарные преобразования с универсальной аппроксимацией при дополнительных условиях плотности.
- Для нейронных сетей главный вывод состоит в том, что подходящие нелинейные активации вместе с линейными комбинациями и суперпозициями дают теоретическую основу для приближения широкого класса функций.

1.11. Проверочные вопросы

1. Какой вопрос о выразительности нейронных сетей ставит теорема Колмогорова?
2. Чем точное представление функции отличается от аппроксимации с заданной точностью?
3. Почему в теореме Колмогорова важен переход от функций многих переменных к функциям одного переменного?
4. Что означает равномерная аппроксимация на компакте?
5. Какие свойства класса функций требуются в теореме Стоуна-Вейерштрасса?
6. Как обобщенная теорема Стоуна связывает нелинейные операции с универсальной аппроксимацией?
7. Почему линейных комбинаций без нелинейности недостаточно для построения богатого класса функций?
8. Какой вывод для нейроинформатики следует из теорем представления и аппроксимации?

Источники

1. Колмогоров А. Н. О представлении непрерывных функций нескольких переменных в виде суперпозиции непрерывных функций

- одного переменного // Доклады АН СССР. 1957. Т. 114, No. 5. С. 953-956.
2. Арнольд В. И. О представлении функций нескольких переменных в виде суперпозиции функций меньшего числа переменных // Математическое просвещение.
 3. Stone M. H. The generalized Weierstrass approximation theorem // Mathematics Magazine. 1948. Vol. 21. P. 167-184, 237-254.
 4. Cybenko G. Approximation by superposition of a sigmoidal function // Mathematics of Control, Signals, and Systems. 1989. Vol. 2. P. 303-314.
 5. Горбань А. Н., Россиев Д. А. Нейронные сети на персональном компьютере. Новосибирск: Наука, 1996.
 6. Иосида К. Функциональный анализ. М.: Мир, 1967.

Лекция 2. Формальное описание искусственных нейронных сетей

Лекция строит формальную модель искусственной нейронной сети через множества входов, выходов, состояний, функции преобразования и матрицы связей. Такой язык нужен, чтобы описывать слоистые, полносвязные, рекуррентные и составные сети в единой математической схеме.

2.1. Постановка проблемы

Перед изучением алгоритмов обучения нужно строго определить объект, который обучается. Нейронная сеть должна быть описана не только рисунком из узлов и стрелок, но и математической конструкцией: какие сигналы поступают на вход, как задаются состояния, как формируются выходы и как связи между нейронами превращаются в матрицы.

2.2. Формальная модель нейрона

Под **формальным нейроном** будем понимать объект

$$F = (S, X, Y, \emptyset, g), \quad 2.1$$

где:

- S - множество состояний нейрона;
- X - множество входных сигналов;
- Y - множество выходных сигналов;
- $\emptyset$ - специальный элемент, обозначающий отсутствие сигнала;
- g - функция преобразования входа и состояния в выходной сигнал.

Обычно функция преобразования задается через взвешенное суммирование входов и нелинейное преобразование:

$$z = \sum_{i=1}^n w_i x_i, \quad 2.2$$

$$y = \varphi(z), \quad 2.3$$

где:

- n - число входов нейрона;

- w_i - вес i -го синапса;
- x_i - i -я компонента входного сигнала;
- z - результат суммирования;
- φ - нелинейное преобразование, или функция активации;
- y - выходной сигнал нейрона.

Такую формулу обычно называют *стандартным формальным нейроном*.

Если объединить несколько нейронов с общим входом, получается слой. Векторная запись ниже относится уже не к одному скалярному нейрону, а к слою из m нейронов:

$$y = \varphi(Wx + b), \quad x \in \mathbb{R}^n, \quad W \in \mathbb{R}^{m \times n}, \quad b \in \mathbb{R}^m, \quad y \in \mathbb{R}^m. \quad 2.4$$

Здесь матрица W задает линейную часть преобразования всего слоя, вектор b задает смещения нейронов слоя, а функция активации φ применяется покомпонентно к вектору $Wx + b$. При $m = 1$ эта формула снова описывает один скалярный нейрон.

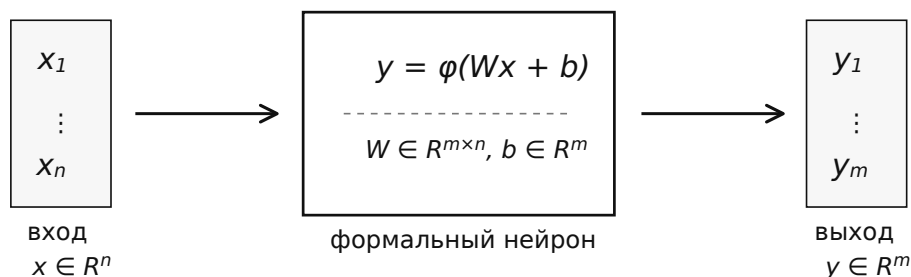


Рис. 2.1. Слой формальных нейронов с векторным входом и выходом.

Формальные нейроны могут рассматриваться в непрерывном или дискретном времени. В дальнейшем будем использовать дискретное время: входной вектор, выходной вектор и состояние нейрона зависят от момента t , но если это не приводит к двусмысленности, параметр времени можно опускать.

Из формулы видно, что простой нейрон реализует скалярное произведение двух векторов, а значит её можно переписать короче:

$$y = \varphi(w \cdot x), \quad 2.5$$

где φ - функция активации, w - вектор весов, x - входной вектор, а $w \cdot x$ - их скалярное произведение.

Скалярное произведение можно также записать через длины векторов и угол между ними:

$$w \cdot x = \|w\| \|x\| \cos \theta. \quad 2.6$$

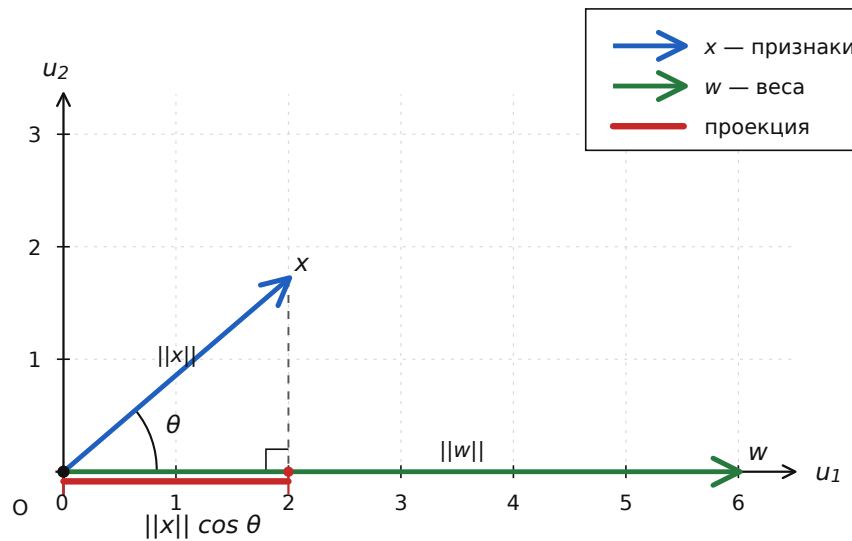


Рис. 2.2. Геометрическая интерпретация скалярного произведения.

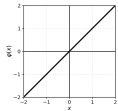
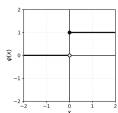
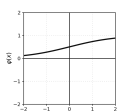
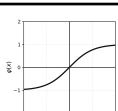
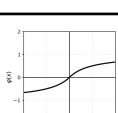
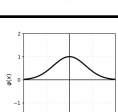
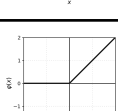
Если после обучения вектор весов w фиксирован, то выражение $w \cdot x$ можно интерпретировать как величину, пропорциональную проекции входного вектора на направление весов. Значение входа активации максимально, когда входной вектор и вектор весов коллинеарны.

В случае линейной функции активации один нейрон реализует линейное преобразование, а значит может быть использован в модели линейной регрессии. В такой модели целью является подбор таких параметров вектора w , которые наилучшим образом аппроксимируют искомую функцию по известным наблюдениям.

2.3. Функции активации

Выбор функции активации ничем не ограничен в общем определении нейрона, но на практике зависит от задачи, алгоритма обучения и вычислительных ресурсов.

Таблица 2.1. Функции активации.

Название	Формула	График	Преимущества	Ограничения
Линейная	$\varphi(x) = x$		Быстрое вычисление, непрерывность, монотонность; производная постоянна и вычисляется без дополнительных затрат	Применима только в некоторых типах сетей
Пороговая	$\varphi(x) = \begin{cases} 1 & x \geq 0 \\ 0 & x < 0 \end{cases}$		Быстрое вычисление, нелинейность	Разрыв в нуле; производная почти всюду равна нулю и не дает полезного градиента
Сигмоидальная	$\varphi(x) = \frac{1}{1+e^{-x}}$		Нелинейность, монотонность, дифференцируемость; производная выражается через уже вычисленное значение $\varphi(x)(1 - \varphi(x))$	Вычисление функции требует экспоненты и дороже простых кусочно-линейных функций
Гиперболический тангенс	$\varphi(x) = \tanh x$		Нелинейность, монотонность, дифференцируемость; производная дешево выражается через уже вычисленное значение $1 - \varphi(x)^2$	Вычисление функции дороже ReLU и линейной функции
Рациональная сигмоида	$\varphi(x) = \frac{x}{1+ x }$		Нелинейность, монотонность, дифференцируемость; производная не требует экспоненты и относительно дешева	В точке смены знака требуется аккуратная обработка модуля
Гауссиан	$\varphi(x) = e^{-x^2}$		Нелинейность, дифференцируемость; производная выражается через значение функции как $-2x\varphi(x)$	Требуется экспоненты, вычисление дороже кусочно-линейных альтернатив
ReLU	$\varphi(x) = \max(0, x)$		Высокая скорость вычисления; производная кусочно постоянна и дешева	Не дифференцируема в нуле; при $x < 0$ производная равна нулю

2.4. Связь двух нейронов

Пусть есть два формальных нейрона F_1 и F_2 . Если i -я компонента выходного вектора нейрона F_1 подается на j -ю компоненту входного вектора нейрона F_2 , то говорят, что соответствующий выход F_1 соединен с соответствующим входом F_2 .

Связь удобно задавать бинарной матрицей. Пусть:

- Y^{F_1} - множество выходов нейрона F_1 ;
- X^{F_2} - множество входов нейрона F_2 ;
- $m = |Y^{F_1}|$;
- $n = |X^{F_2}|$.

Тогда матрица связи

$$M_{F_1 F_2} \in \{0, 1\}^{m \times n} \quad 2.7$$

имеет элемент 1 там, где соответствующий выход первого нейрона связан с соответствующим входом второго, и 0 в остальных случаях.

Если $y^{F_1}(t)$ - выходной вектор первого нейрона, то входной вклад в F_2 можно записать как

$$x^{F_2}(t) = y^{F_1}(t) M_{F_1 F_2}. \quad 2.8$$

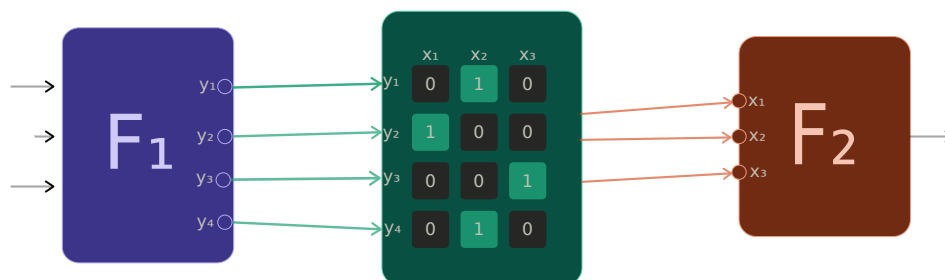


Рис. 2.3. Связь выходов одного формального нейрона со входами другого.

Для двух несвязанных элементов матрица связи является нулевой.

2.5. Матрица внутренних связей множества нейронов

Пусть задано множество нейронов

$$E = \{F_1, F_2, \dots, F_k\}. \quad 2.9$$

Для каждой пары нейронов можно определить матрицу связи M_{ij} . Тогда матрицу внутренних связей всего множества E удобно записывать как блочную матрицу

$$M = \begin{pmatrix} M_{11} & M_{12} & \dots & M_{1k} \\ M_{21} & M_{22} & \dots & M_{2k} \\ \vdots & \vdots & \ddots & \vdots \\ M_{k1} & M_{k2} & \dots & M_{kk} \end{pmatrix}. \quad 2.10$$

Если отдельные блоки имеют разные размеры, их можно дополнять нулями справа и снизу. Это соответствует добавлению фиктивных входов и выходов функциональным элементам.

Для всех нейронов множества E можно рассматривать:

- общий входной вектор x ;
- общий вектор состояния s ;
- общий вектор выходов y .

2.6. Определение искусственной нейронной сети

Искусственная нейронная сеть - это структура, состоящая из связанных между собой формальных нейронов. Формально ее можно задать как объект

$$N = (E, G, H, M, x_0, s_0, T), \quad 2.11$$

где:

- E - множество формальных нейронов;
- G - вектор-функция сети, определяющая общий выходной вектор по общему входному вектору и вектору состояния;
- H - функция изменения состояния сети;
- M - матрица внутренних связей множества E ;
- x_0 - начальный входной вектор;
- s_0 - начальный вектор состояния;
- T - критерий останова функционирования сети, в простейшем случае число тактов.

Большинство современных моделей рассматривает нейронные сети, функционирующие в дискретном времени.

2.7. Универсальная аппроксимация нейронными сетями

Утверждение. Для компактного множества $K \subset \mathbb{R}^n$ и непрерывной функции $F : K \rightarrow \mathbb{R}^m$ при условиях теоремы универсальной аппроксимации можно построить нейронную сеть, которая приближает F с любой заданной точностью в равномерной норме.

Это утверждение важно для прикладных задач: если решение задачи можно представить как построение некоторой непрерывной функции на ограниченной области, то теоретически можно построить нейронную сеть, приближающую эту функцию. Условия на компакт, норму и функцию активации нельзя опускать: без них утверждение становится слишком широким.

Класс функций, вычисляемых нейронными сетями, замкнут относительно линейных операций: если несколько сетей вычисляют функции $F_1, \dots, F_k$, то их линейная комбинация

$$a_0 + a_1 F_1 + \dots + a_k F_k \quad 2.12$$

может быть вычислена сетью с сумматором на выходе.

Класс функций, вычисляемых нейронными сетями, также замкнут относительно унарной нелинейной операции, реализуемой функцией активации. Если сеть вычисляет функцию F , то добавление нелинейного преобразователя дает функцию $\varphi(F)$.

Доказательство этих двух свойств конструктивно.

Для линейной комбинации достаточно взять сети, вычисляющие функции $F_1, \dots, F_k$, и подать их выходы на сумматор с весами $a_1, \dots, a_k$. Тогда на выходе получится функция

$$a_0 + a_1 F_1 + \dots + a_k F_k. \quad 2.13$$

Если отдельные сети выдают ответ за разное число тактов, эту разницу можно компенсировать линиями задержки, составленными из связей с единичным весом.

Для нелинейного преобразования достаточно взять сеть, вычисляющую функцию F , и подать ее выход на нейрон с функцией активации φ . Тогда на выходе новой сети будет функция $\varphi(F)$.

Следовательно, класс функций, реализуемых нейронными сетями, замкнут относительно линейных комбинаций и применения непрерыв-

ной нелинейной функции активации. Вместе с теоремой из предыдущей лекции о плотности такого класса в пространстве непрерывных функций это и дает доказательство утверждения об универсальной аппроксимации.

Теорема. При условиях соответствующей теоремы универсальной аппроксимации множество функций, вычисляемых нейронными сетями с заданной нелинейной функцией активации, плотно в пространстве непрерывных функций на компакте.

2.8. Соединение двух ИНС

Пусть заданы две нейронные сети N_1 и N_2 с множествами нейронов E_1 и E_2 , а также матрицами внутренних связей M_{11} и M_{22} .

Следующая схема иллюстрирует соединение двух ИНС и наличие внешних матриц связей M_{12} и M_{21} :

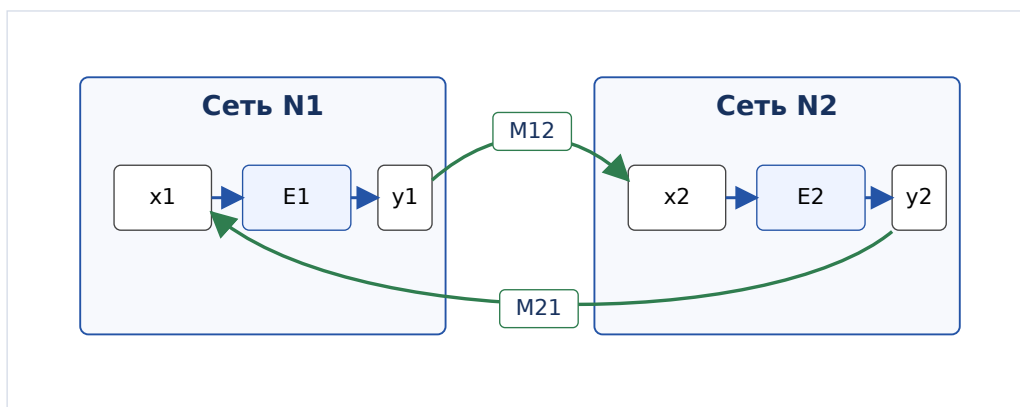


Рис. 2.4. Соединение двух искусственных нейронных сетей через внешние матрицы связей.

Новую сеть N можно определить как соединение этих сетей:

$$E = E_1 \cup E_2. \quad 2.14$$

Матрица внутренних связей новой сети имеет блочный вид

$$M = \begin{pmatrix} M_{11} & M_{12} \\ M_{21} & M_{22} \end{pmatrix}, \quad 2.15$$

где:

- M_{12} задает связи элементов сети N_1 с элементами сети N_2 ;
- M_{21} задает связи элементов сети N_2 с элементами сети N_1 ;
- M_{11} и M_{22} остаются внутренними связями исходных сетей.

Если внешние блоки M_{12} и M_{21} нулевые, функции выхода и изменения состояния новой сети можно строить покомпонентно:

$$G((s_1, s_2), (x_1, x_2)) = (G_1(s_1, x_1), G_2(s_2, x_2)), \quad 2.16$$

$$H((s_1, s_2), (x_1, x_2)) = (H_1(s_1, x_1), H_2(s_2, x_2)). \quad 2.17$$

В общем случае ненулевые матрицы M_{12} и M_{21} добавляют к входам каждой подсети сигналы от другой подсети, поэтому аргументы G_1, G_2, H_1, H_2 должны включать соответствующие внешние вклады. Схема на рисунке 2.4 показывает именно этот более общий случай.

Критерий останова T может определяться по-разному. В простых случаях используют

$$T = T_1 + T_2 \quad 2.18$$

или

$$T = \max(T_1, T_2). \quad 2.19$$

2.9. Многослойные ИНС

Соединение можно обобщить на три и более сетей. Особый класс таких соединений - **многослойные нейронные сети**.

Типичная многослойная сеть состоит из последовательно соединенных слоев:

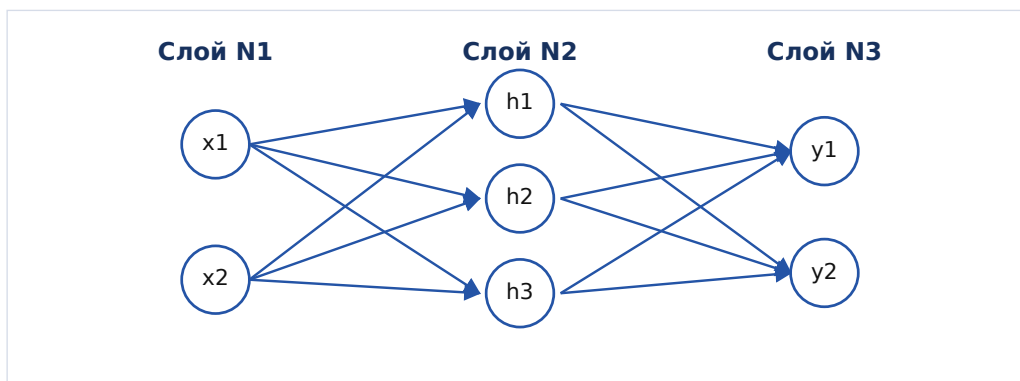


Рис. 2.5. Многослойная искусственная нейронная сеть с последовательными связями между слоями.

Пусть сеть получается последовательным соединением слоев $N_1, \dots, N_k$. Внутри каждого слоя нейроны не связаны между собой, а связи идут только от слоя i к слою $i+1$. Тогда матрица внутренних связей имеет блочно-верхнетреугольный вид:

$$M = \begin{pmatrix} 0 & M_{12} & 0 & \cdots & 0 \\ 0 & 0 & M_{23} & \cdots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & \cdots & M_{k-1,k} \\ 0 & 0 & 0 & \cdots & 0 \end{pmatrix}. \quad 2.20$$

В такой матрице ненулевыми являются только блоки, расположенные непосредственно над главной диагональю. В литературе сеть с такой структурой часто называют **многослойным персептроном**.

Обычно каждый выходной сигнал i -го слоя подается на вход всех нейронов слоя $i + 1$. Число нейронов в разных слоях может быть различным.

2.10. Прямое произведение ИНС

Если несколько сетей соединяются с нулевыми матрицами внешних связей, получается сеть, в которой компоненты функционируют независимо.

На схеме ниже три сети образуют прямое произведение: каждая часть работает независимо от остальных.

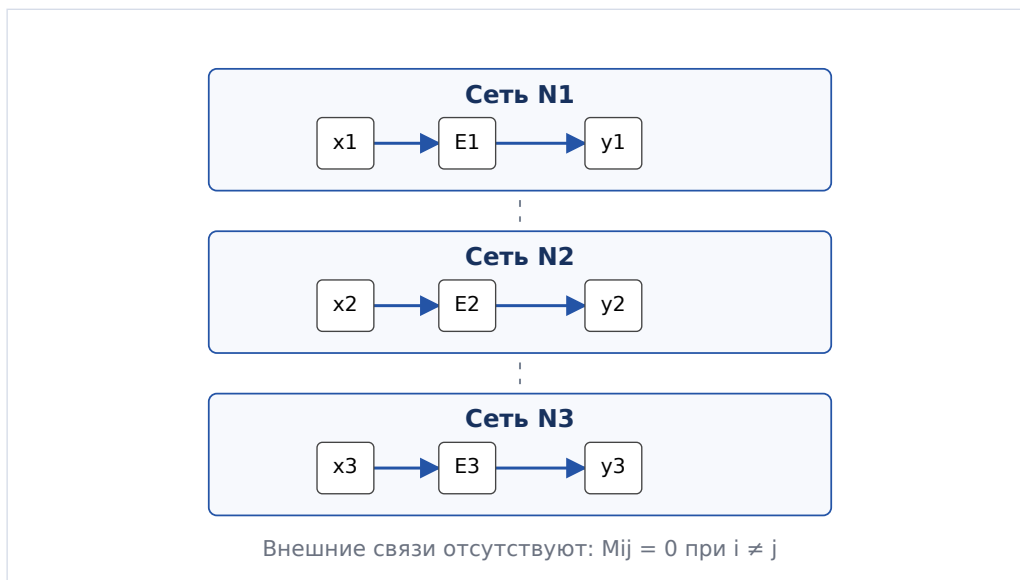


Рис. 2.6. Прямое произведение независимых искусственных нейронных сетей.

Пусть заданы сети $N_1, \dots, N_m$. Для прямого произведения матрица внутренних связей имеет блочно-диагональный вид:

$$M = \begin{pmatrix} M_{11} & 0 & \dots & 0 \\ 0 & M_{22} & \dots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \dots & M_{mm} \end{pmatrix}. \quad 2.21$$

Ненулевые блоки находятся только на главной диагонали. Каждая составляющая сеть функционирует независимо, а результат работы общей сети состоит из отдельных результатов этих сетей.

Если критерий останова задается числом тактов, то для такой сети можно использовать, например,

$$T = \max(T_1, \dots, T_m). \quad 2.22$$

2.11. Полносвязные, однородные и рекуррентные сети

Полносвязная сеть - нейронная сеть, в которой каждый нейрон связан со всеми нейронами этой сети, включая самого себя. Все входные сигналы подаются всем нейронам, а выходными сигналами могут быть все или некоторые выходы нейронов после нескольких тактов функционирования.

Существенное различие между полносвязной и слоистой сетями проявляется, когда число тактов функционирования заранее не ограничено: слоистая сеть в таком режиме работать не может, потому что ее структура задает движение сигнала по слоям.

Однородная сеть - сеть, в которой функция активации нейронов одна и та же для всех нейронов.

Неоднородная сеть - сеть, в которой функция активации зависит от одного или нескольких параметров, значения которых меняются от нейрона к нейрону.

Рекуррентные связи - связи, порождающие зависимость выхода нейронов от значений выхода на предыдущих тактах функционирования.

Рекуррентная ИНС - нейронная сеть с рекуррентными внутренними связями.

Пример рекуррентной сети с обратной связью по скрытому состоянию:

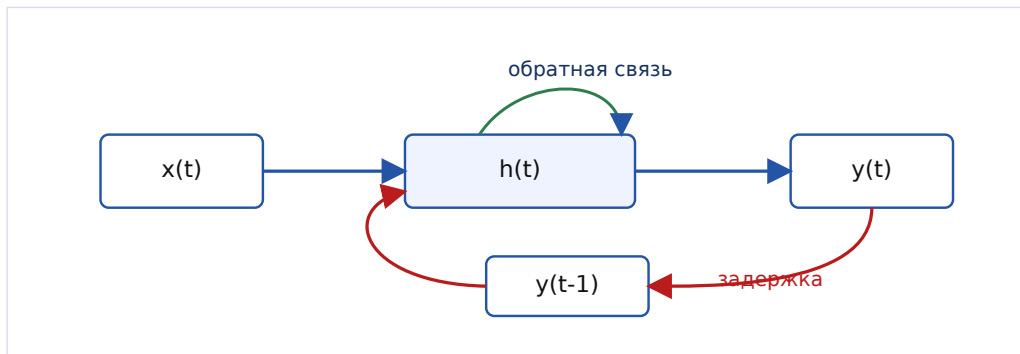


Рис. 2.7. Рекуррентная сеть с обратной связью через задержанный выход.

2.12. Итог

- Формальный нейрон задается множествами входов, выходов, состояний и функцией преобразования.
- Связи между нейронами удобно описывать бинарными матрицами.
- Искусственная нейронная сеть задается множеством нейронов, функциями выхода и состояния, матрицей связей, начальными условиями и критерием останова.
- Сети могут соединяться в более крупные сети через блочные матрицы связей.
- Многослойные, прямые произведения, полносвязные и рекуррентные сети являются частными случаями общей формальной схемы.
- Нелинейная функция активации вместе с линейными операциями и суперпозициями объясняет универсальную аппроксимирующую способность нейронных сетей.

2.13. Проверочные вопросы

1. Какие компоненты входят в формальную модель нейрона?
2. Чем входные, выходные и внутренние связи отличаются в формальном описании сети?
3. Как матрица связей задает структуру взаимодействия нейронов?
4. Почему функция активации является частью определения нейрона?
5. Как формально описывается соединение двух искусственных нейронных сетей?
6. Чем многослойная сеть отличается от полносвязной сети в терминах связей?
7. Что означает прямое произведение искусственных нейронных сетей?

8. Почему рекуррентные связи требуют учитывать такты функционирования?
9. Как формальная схема помогает сравнивать разные архитектуры нейронных сетей?

Источники

1. Митрофанов Ю. И. Системный анализ: учеб. пособие. Саратов: Научная книга, 2000.
2. Богомоллов А. М., Салий В. Н. Алгебраические основы теории дискретных систем. М.: Наука, 1997.
3. Тархов Д. А. Нейронные сети. Модели и алгоритмы. М.: Радиотехника, 2005.
4. Горбань А. Н. Обучение нейронных сетей. М.: СССР-США СП «Параграф», 1990.
5. Головкин В. А. Нейронные сети: обучение, организация и применение. М.: ИПРЖР, 2001.

Лекция 3. Вычисления на графе

В предыдущей лекции связи между функциональными элементами задавались при помощи двоичной матрицы связей. Если все входы и выходы нейросети проиндексировать последовательно, то матрицу связей такой нейросети мы можем считать матрицей смежности графа с соответствующими вершинами.

3.1. Постановка проблемы

Нейронную сеть удобно рассматривать как вычислительный граф: вершины хранят значения, дуги задают направление передачи сигналов, а матрицы описывают структуру связей и веса. Такая запись нужна, чтобы перейти от статического описания сети к пошаговым вычислениям, итерациям и дальнейшему выводу алгоритмов обучения.

3.2. Граф вычислений

Ориентированным графом называется пара (V, E) , где V - множество вершин, а E - множество дуг, соединяющих вершины.

Представление порядка вычислений в виде ориентированного графа удобно для описания того, как сигнал распространяется по нейронной сети и как из значений в одних вершинах получают значения в других. Вершины графа соответствуют локальным переменным или функциональным элементам, а дуги задают зависимости между ними.

Если в вершинах графа стоят функции, то ориентированные дуги определяют порядок их композиции. Поэтому один и тот же ориентированный граф можно рассматривать как компактную запись вычислительной схемы.

Принципиально важным является порядок заходящих в вершину дуг. Если функция в вершине графа чувствительна к порядку, то заходящие в вершину дуги должны быть упорядочены, например, проиндексированы порядковыми номерами.

3.3. Структура графа и матрица смежности

Рассмотрим граф с вершинами

$$V = \{v_1, v_2, v_3, v_4\}. \quad 3.1$$

Связи между вершинами задаются матрицей смежности

$$M = \begin{pmatrix} 0 & 1 & 1 & 0 \\ 1 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \end{pmatrix}. \quad 3.2$$

Из этой матрицы видно, что:

- вершина v_1 передает сигнал в v_2 и v_3 ;
- вершина v_2 передает сигнал в v_1 и v_3 ;
- вершина v_3 передает сигнал в v_4 ;
- вершина v_4 не имеет исходящих дуг.

На рисунке показана структура графа вычислений.

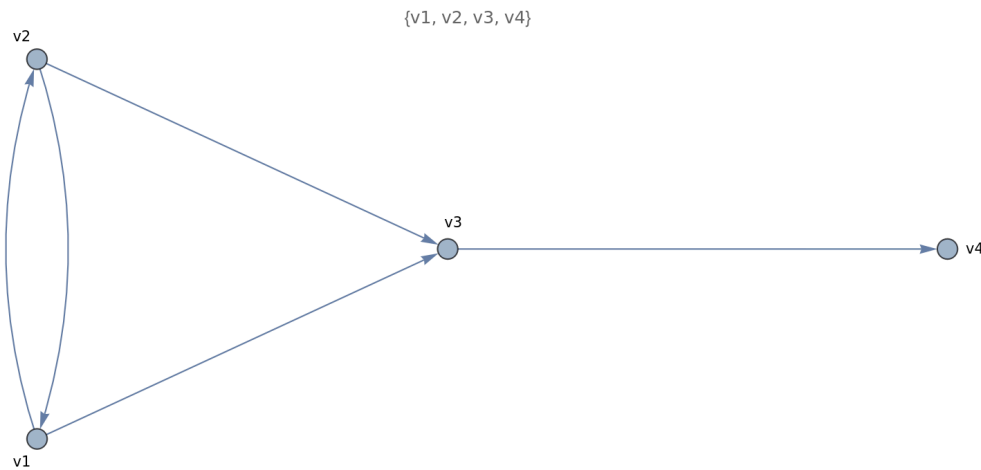


Рис. 3.1. Граф вычислений с метками функций в вершинах.

3.4. Числовой пример

Припишем каждой вершине функцию стандартного формального нейрона $g(x) = f\left(\sum_{i=1}^n y_i A_{ij}\right)$, а каждой дуге графа припишем действительное значение, которое далее будем называть весом дуги или весом связи.

Возьмем начальные значения

$$y^{(0)} = (1, -0.1, 1.5, 0.5) \quad 3.3$$

и зададим веса

$$w_{12} = -1, \quad w_{13} = -1, \quad w_{21} = 1, \quad w_{23} = 1, \quad w_{34} = 1. \quad 3.4$$

Будем использовать соглашение: строка i матрицы соответствует вершине-источнику v_i , столбец j — вершине-приемнику v_j , а элемент A_{ij} задает вес дуги $v_i \rightarrow v_j$. Тогда эффективная матрица связей равна покомпонентному произведению бинарной матрицы смежности M и матрицы весов W :

$$A = M \circ W = \begin{pmatrix} 0 & -1 & -1 & 0 \\ 1 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \end{pmatrix}. \quad 3.5$$

Если текущие значения вершин записаны строковым вектором y , то вход вершины v_j после передачи сигналов равен

$$x_j = \sum_{i=1}^4 y_i A_{ij}, \quad 3.6$$

а весь входной вектор получается умножением справа:

$$x = yA. \quad 3.7$$

В качестве функции активации используем рациональную сигмоиду

$$f(z) = \frac{z}{1 + |z|}. \quad 3.8$$

Тогда после умножения на матрицу связей получаем

$$x^{(1)} = (-0.1, -1, -1.1, 1.5). \quad 3.9$$

После покомпонентного применения функции активации имеем

$$y^{(1)} = (-0.0909, -0.5, -0.5238, 0.6). \quad 3.10$$

Этот результат совпадает с прямым вычислением по графу: каждая вершина получает сумму по входящим дугам, а затем преобразует ее функцией активации.

На рисунке ниже показаны значения вершин после одной числовой итерации.

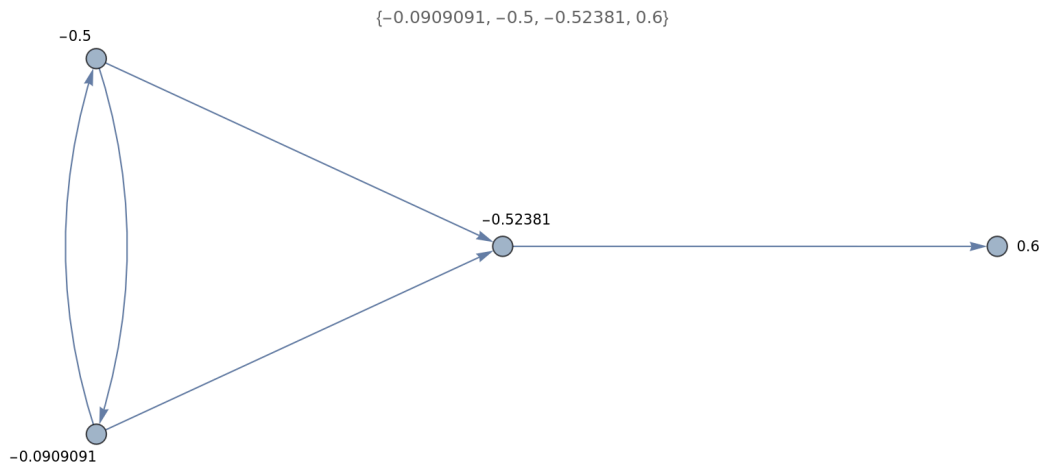


Рис. 3.2. Граф вычислений с метками функций после одной числовой итерации.

3.5. Итерационные вычисления на графе

На практике, при фиксированной матрице M , веса задаются только для ненулевых связей, а значит произведение $A = M \circ W = W$. Поэтому матрица M в вычислениях подразумевается, но не пишется. В других случаях, когда маршрутизация между элементами нейросети не фиксирована, M может меняться, и поэтому участвует в вычислениях наравне с W .

Если граф содержит циклы, вычисление можно повторять много раз. Тогда возникает рекуррентный процесс

$$y^{(n)} = f(y^{(n-1)}A), \quad n = 1, 2, \dots \quad 3.11$$

В исходном вычислительном эксперименте этот процесс задавался как таблица рекуррентных значений на нескольких десятках шагов. Такой режим особенно важен для рекуррентных нейронных сетей, где текущее состояние зависит от предыдущего.

Итерационный анализ позволяет исследовать:

- стабилизацию значений в вершинах;
- затухание сигнала;
- рост значений при неудачном выборе коэффициентов;
- влияние функции активации на динамику сети.

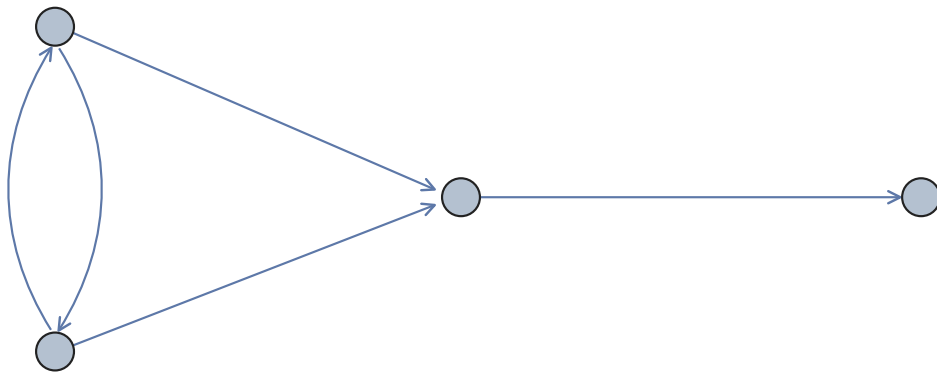


Рис. 3.3. Несколько итераций числового вычисления на графе.

3.6. Связь с нейронными сетями

Рассмотренная схема непосредственно связана с моделью функционирования нейронной сети. Вектор y можно трактовать как выходы нейронов на текущем такте, матрицу $A = M \circ W$ - как структуру связей с весами, а отображение $y_{t+1} = f(y_t A)$ как один шаг распространения сигнала по сети.

В таком подходе граф вычислений полезен по двум причинам:

- он наглядно показывает архитектуру зависимостей между нейронами;
- он позволяет записывать вычисления в матричном виде, удобном для анализа и программной реализации.

3.7. Итог

- Граф вычислений задает порядок передачи сигналов и композиции локальных преобразований.
- Матрица смежности описывает структуру дуг, а матрица весов задает численные коэффициенты связей.
- Один шаг вычислений на графе записывается как умножение вектора текущих значений на эффективную матрицу связей с последующим применением функции активации.
- При наличии циклов граф естественно переходит к итерационной, то есть рекуррентной, схеме вычислений.

3.8. Проверочные вопросы

1. Как матрица смежности описывает ориентированный граф вычислений?
2. Чем матрица весов отличается от бинарной матрицы смежности?
3. Как записывается одношаговое обновление значений вершин графа?
4. Почему порядок индексации вершин важен при построении матриц графа?
5. Как интерпретировать нулевой элемент в матрице смежности?
6. Что меняется при переходе от одной итерации вычислений к нескольким?
7. Как граф вычислений связан с формальной моделью нейронной сети?
8. Почему графовая запись удобна для дальнейшего вывода алгоритмов обучения?

Источники

1. Тархов Д. А. Нейронные сети. Модели и алгоритмы. М.: Радиотехника, 2005.
2. Goodfellow I., Bengio Y., Courville A. [Deep Learning. Chapter 6: Deep Feedforward Networks](#). MIT Press, 2016.
3. Baydin A. G., Pearlmutter B. A., Radul A. A., Siskind J. M. [Automatic differentiation in machine learning: a survey](#). JMLR, 2018.

Лекция 4. Задача обучения нейронных сетей

Лекция переводит обучение нейронной сети на язык многомерной оптимизации. В ней вводятся обучающая выборка, эмпирическая функция потерь, пространство параметров, итерационные методы и причины, по которым оптимизация глубоких сетей оказывается трудной.

4.1. Постановка проблемы

Пусть задана нейронная сеть

$$N = (E, G, H, M, x_0, s_0, T), \quad 4.1$$

где архитектура сети фиксирована: известны множество нейронов, структура связей, функции выхода и правило функционирования. При фиксированной архитектуре изменяемыми величинами остаются параметры сети: синаптические веса, смещения и, если это допускается моделью, параметры функций активации.

Обозначим вектор всех настраиваемых параметров через

$$\theta = (\theta_1, \dots, \theta_d) \in \mathbb{R}^d. \quad 4.2$$

Тогда сеть реализует параметрическое семейство функций

$$f_\theta : X \rightarrow Y. \quad 4.3$$

Обучение сети состоит в выборе такого вектора параметров θ , при котором функция f_θ наилучшим образом приближает требуемое отображение.

4.2. Обучающая выборка и целевая функция

Пусть желаемый результат обучения задается функцией

$$R : X \rightarrow Y. \quad 4.4$$

На практике функция R обычно известна не полностью, а через конечный набор примеров.

Обучающая выборка - это набор пар

$$\mathcal{D} = \{(x^{(m)}, y^{(m)})\}_{m=1}^M, \quad 4.5$$

где

$$y^{(m)} = R(x^{(m)}). \quad 4.6$$

Чтобы сравнить ответ сети с правильным ответом, вводят функцию ошибки, или функцию потерь:

$$\ell(f_\theta(x), y). \quad 4.7$$

Она показывает, насколько выход сети $f_\theta(x)$ отличается от целевого значения y на одном примере.

На всей обучающей выборке вводят функционал ошибки:

$$D_{\mathcal{D}}(\theta) = \frac{1}{M} \sum_{m=1}^M \ell(f_\theta(x^{(m)}), y^{(m)}). \quad 4.8$$

Функционал $D_{\mathcal{D}}$ называют также целевой функцией обучения или эмпирическим риском.

4.3. Формальная задача многомерной оптимизации

При фиксированной архитектуре задача обучения нейронной сети формально сводится к задаче многомерной оптимизации:

$$\theta^* \in \arg \min_{\theta \in \mathbb{R}^d} D_{\mathcal{D}}(\theta). \quad 4.9$$

Здесь:

- θ - точка в многомерном пространстве параметров;
- d - число обучаемых параметров сети;
- $D_{\mathcal{D}}(\theta)$ - функция многих переменных;
- θ^* - набор параметров, при котором ошибка на обучающей выборке минимальна.

Если вместо конечной выборки доступно распределение данных, можно записать теоретический риск:

$$D(\theta) = \mathbb{E}_{(x,y)}[\ell(f_\theta(x), y)], \quad 4.10$$

а задача обучения принимает вид

$$\theta^* \in \arg \min_{\theta \in \mathbb{R}^d} D(\theta). \quad 4.11$$

На практике обычно минимизируют эмпирический риск $D_{\mathcal{D}}(\theta)$, потому что истинное распределение данных неизвестно.

4.4. Частичная и полная задача обучения

В этой постановке рассматривается **частичная задача обучения**: архитектура сети задана заранее, а подобрать нужно только параметры θ .

Если же неизвестны число нейронов, типы нейронов, структура связей и функции активации, задача становится шире. Тогда нужно выбирать не только точку θ в пространстве параметров, но и саму архитектуру. Такая задача сложнее: она сочетает дискретный выбор структуры и непрерывную оптимизацию параметров.

Поэтому удобно разделять:

- **оптимизацию параметров** при заданной архитектуре;
- **поиск архитектуры**, где меняется само семейство функций f_θ .

Первое обычно является нелинейной задачей оптимизации в пространстве параметров или на выбранном допустимом множестве параметров, а последнее - дискретной задачей оптимизации.

4.5. Обучение как невыпуклая многоэкстремальная задача

Функция ошибки нейронной сети обычно имеет сложный вид. Причины:

- функция f_θ задается суперпозицией многих линейных и нелинейных преобразований;
- параметры входят в вычисления нелинейно;
- размерность пространства параметров может быть очень большой;
- обучающая выборка задает много локальных требований к функции;
- разные наборы параметров могут давать близкие значения ошибки.

Поэтому обучение нейронной сети в общем случае является невыпуклой многоэкстремальной задачей оптимизации - $D_{\mathcal{D}}$ может иметь много локальных минимумов, седловых точек и плоских областей.

Итерационные алгоритмы обучения не гарантируют нахождение глобального минимума в общем случае. Их практическая цель - найти достаточно хорошую точку параметров за приемлемое время.

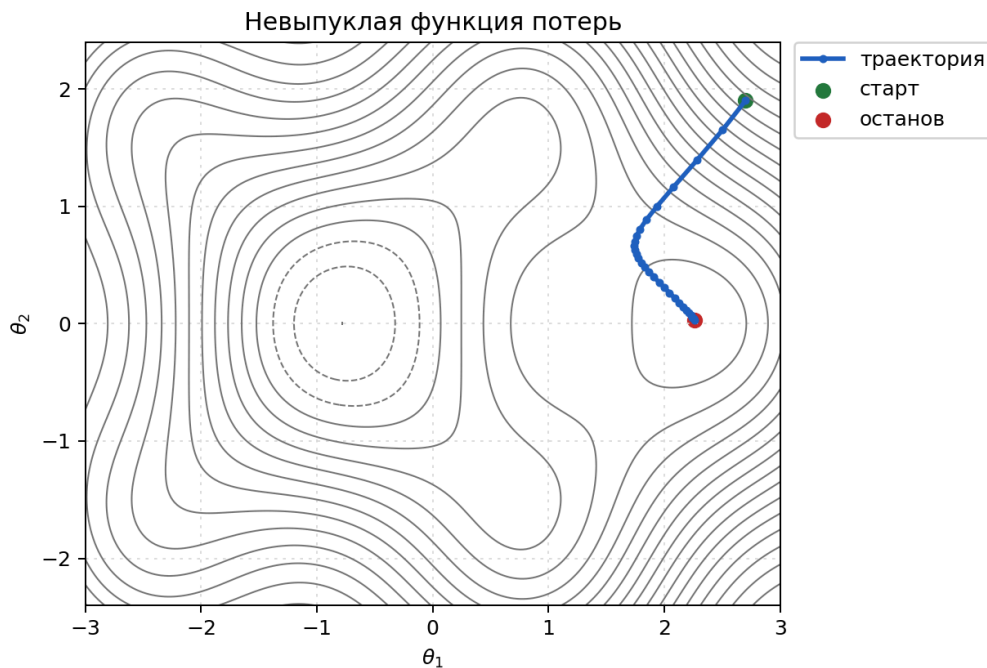


Рис. 4.1. Невыпуклая функция потерь и траектория итерационного спуска.

4.6. Типовые функции потерь

4.6.1. Средняя абсолютная ошибка

Средняя абсолютная ошибка (mean absolute error, MAE) измеряет среднее модульное отклонение предсказаний от целевых значений:

$$\text{MAE}(\theta) = \frac{1}{M} \sum_{m=1}^M |f_{\theta}(x^{(m)}) - y^{(m)}|. \quad 4.12$$

Геометрически она соответствует усредненному расстоянию в норме L_1 .

MAE обычно применяют в задачах регрессии, когда важно уменьшать среднюю абсолютную ошибку без чрезмерной чувствительности к редким большим выбросам. Эта функция потерь полезна, если шум в данных может быть негауссовским или если большие отклонения не должны доминировать в процессе обучения.

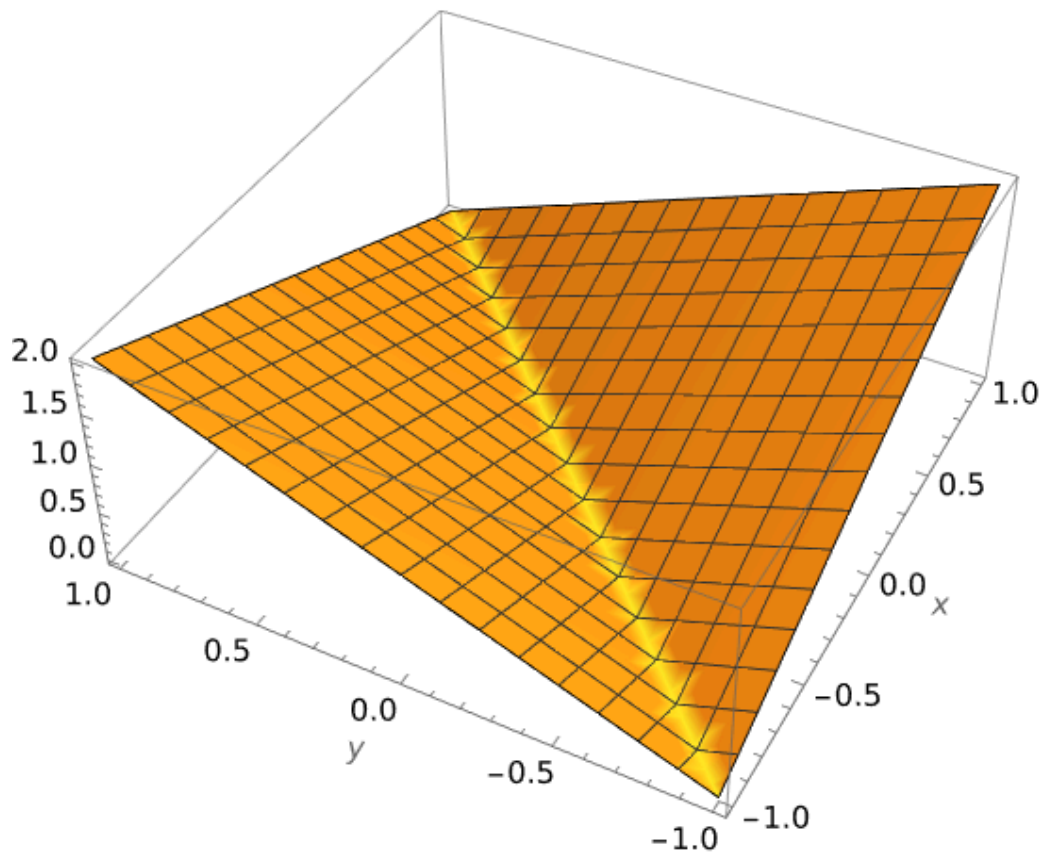


Рис. 4.2. Средняя абсолютная ошибка.

4.6.2. Среднеквадратичная ошибка

Среднеквадратичная ошибка (mean squared error, MSE) измеряет средний квадрат отклонения:

$$\text{MSE}(\theta) = \frac{1}{M} \sum_{m=1}^M (f_{\theta}(x^{(m)}) - y^{(m)})^2. \quad 4.13$$

Для векторных выходов часто используют квадрат евклидовой нормы:

$$\text{MSE}(\theta) = \frac{1}{M} \sum_{m=1}^M \|f_{\theta}(x^{(m)}) - y^{(m)}\|_2^2. \quad 4.14$$

MSE чаще всего используют в задачах регрессии и аппроксимации непрерывных зависимостей, когда большие ошибки нужно штрафовать сильнее малых. Она особенно естественна в моделях, где предполагается гауссовский шум наблюдений.

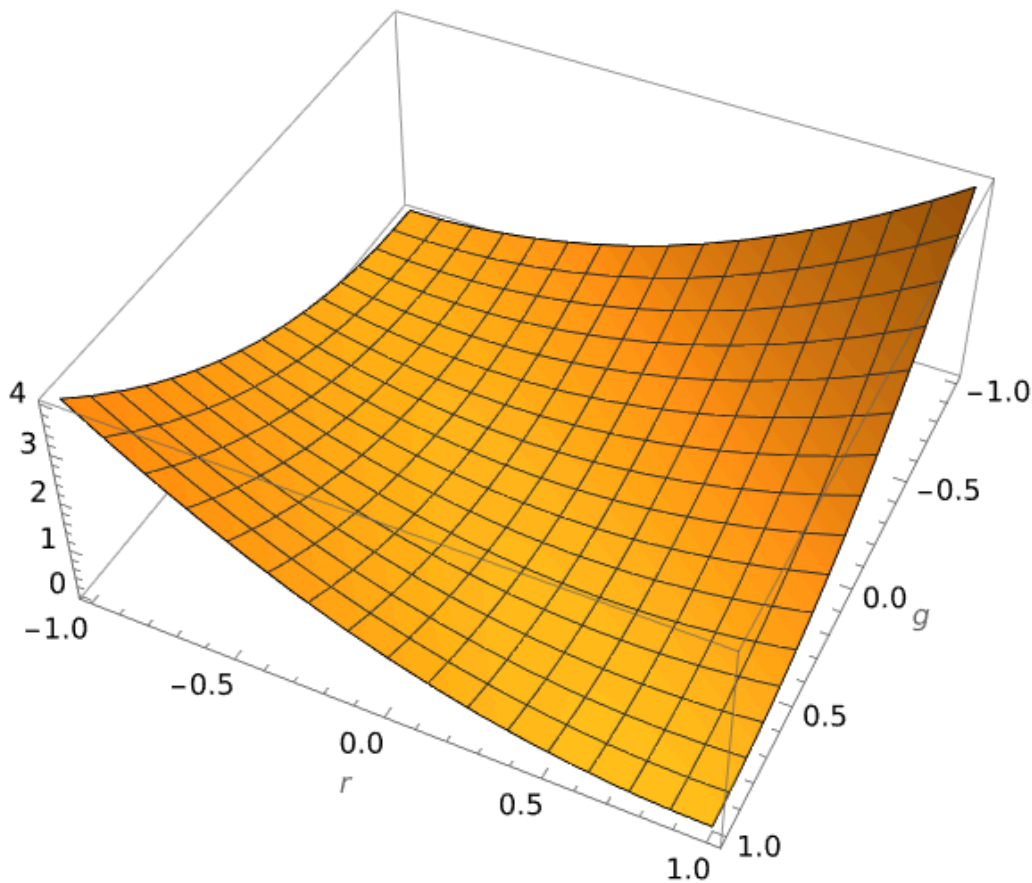


Рис. 4.3. Среднеквадратичная ошибка.

4.6.3. Расстояние Кульбака-Лейблера

Дивергенцию Кульбака-Лейблера (Kullback-Leibler divergence, KL-дивергенция) применяют для сравнения распределений вероятностей:

$$D_{\text{KL}}(p \parallel q) = \sum_i p_i \log \frac{p_i}{q_i}. \quad 4.15$$

Здесь p - целевое распределение, а q - распределение, предсказанное моделью; предполагается, что $p_i \geq 0$, $q_i > 0$ и суммы компонент равны единице. Если $p_i = 0$, соответствующее слагаемое по непрерывному продолжению считают равным нулю.

Дивергенция Кульбака-Лейблера применяется в задачах, где ответ модели имеет вид распределения вероятностей: при сравнении вероятностных прогнозов, в вариационных моделях, при дистилляции знаний и при обучении моделей, согласующих апостериорные и априорные распределения.

4.6.4. Кросс-энтропия

Для бинарной классификации кросс-энтропия имеет вид

$$\ell(z, y) = -(y \log z + (1 - y) \log(1 - z)), \quad 4.16$$

где z - предсказанная вероятность класса 1, а $y \in \{0, 1\}$ - правильная метка.

В этой формуле важно различать вероятность класса 1 и вероятность правильного класса. Модель в бинарной классификации возвращает одно число $z \in (0, 1)$. Это число интерпретируется как вероятность события, что объект относится к классу 1. Тогда вероятность класса 0 равна $1 - z$.

Если правильная метка равна $y = 1$, то вероятность правильного класса равна z , и кросс-энтропия принимает вид

$$\ell(z, 1) = -\log z. \quad 4.17$$

Если правильная метка равна $y = 0$, то правильным является класс 0, поэтому используется не z , а вероятность $1 - z$:

$$\ell(z, 0) = -\log(1 - z). \quad 4.18$$

Иными словами, бинарная кросс-энтропия всегда штрафует модель за малую вероятность, назначенную правильному классу. Если модель уверенно выбирает неправильный класс, то вероятность правильного класса мала, а значение $-\log p$ становится большим. Например, при $y = 0$ и $z = 0.75$ модель назначает классу 1 вероятность 0.75, но правильному классу 0 остается вероятность $1 - z = 0.25$. Поэтому потеря равна $-\log 0.25 = \log 4$.

Для определения многоклассовой классификации нам понадобится понятие логитов. Пусть нейросеть для объекта x выдает вектор

$$a_\theta(x) = (a_1(x), \dots, a_C(x)) \in \mathbb{R}^C. \quad 4.19$$

Число $a_c(x)$ называют логитом класса c . Логит - это ненормированная оценка класса до преобразования к норме вероятностного пространства: он может принимать любое вещественное значение, а его относительная величина показывает, насколько модель предпочитает один класс другим.

Для классификации удобно использовать не сам вектор логитов, а нормированный вектор

$$p_\theta(x) = (p_{\theta,1}(x), \dots, p_{\theta,C}(x)), \quad p_{\theta,c}(x) \geq 0, \quad \sum_{c=1}^C p_{\theta,c}(x) = 1. \quad 4.20$$

Каждый элемент вектора трактуется как оценка модели соответствующего класса. Благодаря свойствам неотрицательности и единич-

ности суммы компонента $p_{\theta,c}(x)$ может интерпретироваться как предсказанная вероятность того, что объект x относится к классу c . В обычной многоклассовой классификации классы взаимоисключающие: объект имеет одну правильную метку, а вероятности по всем классам суммируются в единицу.

Для многоклассовой классификации правильная метка имеет вид $y \in \{1, \dots, C\}$, а кросс-энтропия берет отрицательный логарифм вероятности правильного класса:

$$\ell(f_{\theta}(x), y) = -\log p_{\theta,y}(x). \quad 4.21$$

На практике вектор вероятностей часто получают из логитов функцией нормированной экспоненты (softmax):

$$p_{\theta,c}(x) = \frac{\exp(a_c(x))}{\sum_{j=1}^C \exp(a_j(x))}. \quad 4.22$$

Кросс-энтропию обычно используют в задачах классификации, где модель должна предсказывать вероятности классов. Для бинарной классификации она естественно сочетается с сигмоидальным выходом, а для многоклассовой - с softmax, поэтому именно эта функция потерь стала стандартом для большинства задач распознавания и категоризации.

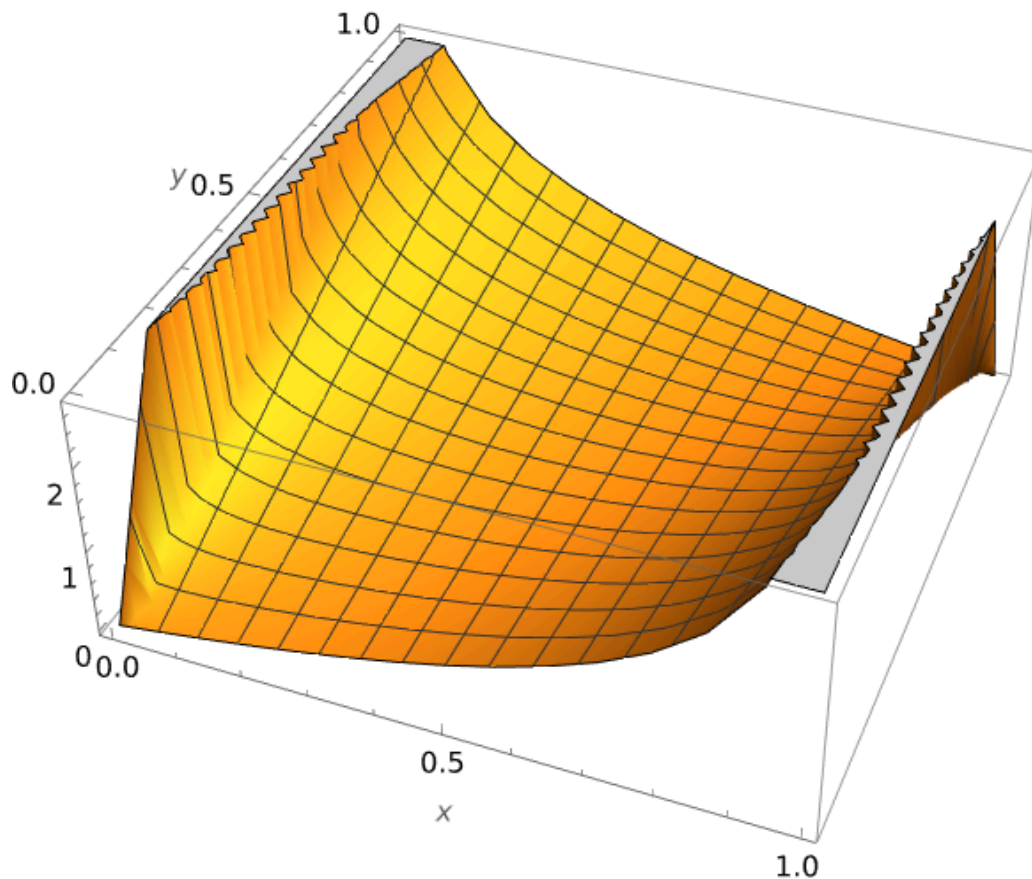


Рис. 4.4. Кросс-энтропия.

4.7. Данные для обучения

Формальная запись выборки $\mathcal{D}$ скрывает важный практический этап: перед обучением данные нужно привести к виду, в котором функция потерь действительно отражает решаемую задачу. Если в выборке есть ошибки разметки, пропуски или сильно отличающиеся масштабы признаков, оптимизатор может уменьшать не содержательную ошибку модели, а артефакты подготовки данных.

Предобработка обучающей выборки обычно включает несколько шагов.

1. **Обработка выбросов.** Аномальные наблюдения проверяют: их удаляют, исправляют или оставляют с явной пометкой, если они являются редкими, но реальными случаями.
2. **Заполнение пропусков.** Пропущенные значения заменяют статистиками по обучающей выборке, специальными категориями, модельными оценками или масками пропусков.
3. **Нормализация признаков.** Числовые признаки приводят к сопоставимым масштабам, например через центрирование и деление

на стандартное отклонение. Это особенно важно для методов, чувствительных к масштабу координат.

4. **Аугментация данных.** Для некоторых типов данных создают дополнительные допустимые варианты объектов: повороты и сдвиги изображений, шумовые возмущения сигналов, переформулировки текстов. Аугментация должна сохранять правильный ответ y и применяться только к обучающей части данных.

Для контроля качества обобщения выборку обычно делят на три непересекающиеся подвыборки:

$$\mathcal{D} = \mathcal{D}_{\text{train}} \cup \mathcal{D}_{\text{val}} \cup \mathcal{D}_{\text{test}}, \quad \mathcal{D}_i \cap \mathcal{D}_j = \emptyset \quad (i \neq j). \quad 4.23$$

Обучающая подвыборка $\mathcal{D}_{\text{train}}$ используется для подбора параметров θ :

$$\theta^* \approx \arg \min_{\theta} D_{\mathcal{D}_{\text{train}}}(\theta). \quad 4.24$$

Валидационная подвыборка $\mathcal{D}_{\text{val}}$ нужна для выбора гиперпараметров: архитектуры, шага обучения, регуляризации, размера минибатча и правила останова. Она помогает сравнивать варианты модели без прямой подгонки под тестовую часть.

Тестовая подвыборка $\mathcal{D}_{\text{test}}$ используется один раз в конце для итоговой оценки качества. Если многократно выбирать модель по тестовой ошибке, тест перестает быть независимой проверкой и фактически превращается в еще одну валидационную выборку.

На практике часто используют пропорции 70:15:15 или 80:10:10, но точное разбиение зависит от размера данных. Для классификации полезно сохранять доли классов в каждой подвыборке. Для временных рядов случайное перемешивание может нарушить причинный порядок, поэтому данные обычно делят по времени: прошлые наблюдения идут в обучение, более поздние - в валидацию и тест.

Важное правило: все статистики предобработки нужно вычислять только по $\mathcal{D}_{\text{train}}$, а затем применять к валидационной и тестовой частям. Иначе информация из будущей проверки попадет в обучение; такую ошибку называют утечкой данных (data leakage).

4.8. Задача аппроксимации функции в стандартной постановке

Рассмотрим задачу аппроксимации в стандартной постановке. Пусть задана функция R , известная через обучающую выборку

$$\mathcal{D} = \{(x^{(m)}, y^{(m)})\}_{m=1}^M. \quad 4.25$$

Нужно найти вектор параметров θ^* такой, что нейронная сеть f_{θ^*} наилучшим образом аппроксимирует функцию R :

$$\theta^* \in \arg \min_{\theta \in \mathbb{R}^d} \frac{1}{M} \sum_{m=1}^M \ell(f_{\theta}(x^{(m)}), y^{(m)}). \quad 4.26$$

Если модель линейна по параметрам, задача иногда сводится к решению системы линейных уравнений. Однако даже для линейных нейросетей данный способ становится неэффективным при числе параметров более 10^3 . Кроме того, поиск минимума через решение систем линейных уравнений может столкнуться со сложностями: вырожденность матрицы коэффициентов, слишком маленькие коэффициенты, нестабильность вычислений. В случае, когда f_{θ} является нелинейной суперпозицией, единственный подходящий класс методов аппроксимации - это поиск оптимума итерационными методами нелинейной оптимизации.

Все это сделало использование классических методов малоприменимыми для задачи поиска оптимальных параметров нейронных сетей. Далее рассмотрим некоторые методы машинного обучения, показавшие свою эффективность в нейроинформатике.

4.9. Стратегии обучения

Обучение с учителем использует выборку пар $(x^{(m)}, y^{(m)})$. Параметры подбираются так, чтобы выход сети был близок к известным правильным ответам.

Обучение без учителя использует набор входных примеров $x^{(m)}$ без заранее заданных правильных ответов. Качество результата оценивается косвенно: через реконструкцию, компактность представлений, энергию модели, кластерную структуру или другие критерии.

Смешанное обучение сочетает оба подхода: часть параметров или этапов обучения использует размеченные данные, а часть работает без явных правильных ответов.

4.10. Итерационный характер обучения

Обучение обычно является итерационной процедурой. На каждом шаге алгоритм имеет текущий вектор параметров θ_k и строит следующий вектор θ_{k+1} так, чтобы уменьшить целевую функцию:

$$D_{\mathcal{D}}(\theta_{k+1}) \leq D_{\mathcal{D}}(\theta_k), \quad 4.27$$

или хотя-бы уменьшить ее в среднем на последовательности шагов.

Общий вид шага оптимизации:

$$\theta_{k+1} = \theta_k + \eta_k p_k, \quad 4.28$$

где:

- p_k - направление движения в пространстве параметров;
- η_k - длина шага, или скорость обучения;
- k - номер итерации.

Выбор направления p_k , шага η_k и правила останова задает конкретный алгоритм обучения.

4.11. Градиентный спуск

Один из базовых методов непрерывной нелинейной оптимизации - градиентный спуск. В нем направление движения выбирается противоположным градиенту функции ошибки:

$$p_k = -\nabla D_{\mathcal{D}}(\theta_k). \quad 4.29$$

Тогда шаг обучения имеет вид

$$\theta_{k+1} = \theta_k - \eta_k \nabla D_{\mathcal{D}}(\theta_k). \quad 4.30$$

Градиент показывает направление наибольшего роста функции ошибки, поэтому движение в направлении $-\nabla D_{\mathcal{D}}(\theta_k)$ локально уменьшает ошибку.

В нейронных сетях градиент обычно вычисляется методом обратного распространения ошибки. После этого параметры обновляются выбранным оптимизационным алгоритмом.

4.12. Обобщенный алгоритм нелинейной оптимизации

Общую схему обучения можно описать так.

1. Выбрать начальные параметры θ_0 , начальный шаг η_0 и правило останова.
2. На k -й итерации вычислить значение функции ошибки $D_{\mathcal{D}}(\theta_k)$.
3. Определить направление движения p_k .
4. Определить шаг η_k .
5. Обновить параметры:

$$\theta_{k+1} = \theta_k + \eta_k p_k. \quad 4.31$$

6. Если ошибка достаточно мала или выполнено правило останова, завершить обучение. Иначе перейти к следующей итерации.

Критериями останова могут быть:

- достижение заданного числа итераций;
- достижение заданного уровня ошибки;
- малое изменение ошибки между итерациями;
- малое изменение параметров;
- отсутствие улучшения на валидационной выборке.

4.13. Классы методов оптимизации

Методы обучения можно классифицировать по информации, которую они используют для выбора направления движения.

Методы нулевого порядка не требуют вычисления градиента. Они используют только значения функции ошибки. Примеры: поиск в случайном направлении, перебор, некоторые стохастические.

Методы первого порядка используют первые производные функции ошибки. Примеры: градиентный спуск, стохастический градиентный спуск, методы с импульсом, Adam, методы сопряженных градиентов.

Методы второго порядка используют информацию о вторых производных или их приближениях. Примеры: метод Ньютона, квазиньютоновские методы, метод Гаусса-Ньютона, метод Левенберга-Марквардта.

Методы более высоких порядков редко используются в обучении нейронных сетей, потому что вычислительные затраты быстро становятся слишком большими.

Методы глобальной оптимизации используют алгоритмы с поиском по всему пространству параметров. К таким методам относятся методы рестартов, метод имитации отжига металлов, генетические алгоритмы.

4.14. Почему оптимизация нейросетей трудна

Основные трудности традиционного подхода к оптимизации нейронных сетей:

1. **Высокая размерность.** Современные сети могут иметь тысячи, миллионы или миллиарды параметров.
2. **Невыпуклость.** Целевая функция часто имеет много локальных минимумов, седловых точек и плато.
3. **Большие вычислительные затраты.** На каждой итерации нужно вычислять выход сети, ошибку и часто градиенты.
4. **Плохая обусловленность.** Разные направления в пространстве параметров могут иметь сильно различную кривизну.
5. **Зависимость от данных.** Качество найденного решения определяется не только оптимизацией, но и обучающей выборкой.
6. **Переобучение.** Минимальная ошибка на обучающей выборке не всегда означает хорошее качество на новых данных.



Рис. 4.5. Основные трудности оптимизации нейронных сетей.

4.15. Итог

- При фиксированной архитектуре обучение нейронной сети - это выбор вектора параметров $\theta \in \mathbb{R}^d$.
- Функция потерь измеряет ошибку на одном примере, а функционал ошибки измеряет качество сети на выборке.
- Формальная задача обучения записывается как $\theta^* \in \arg \min_{\theta \in \mathbb{R}^d} D_{\mathcal{D}}(\theta)$.
- Качество обучения зависит от подготовки данных: обработки выбросов, пропусков, масштабов признаков, аугментации и корректного разделения на обучающую, валидационную и тестовую части.
- В общем случае это многомерная невыпуклая многоэкстремальная задача оптимизации.
- Основным практический подход - итерационное уменьшение ошибки через шаги вида $\theta_{k+1} = \theta_k + \eta_k p_k$.
- Градиентный спуск выбирает направление $p_k = -\nabla D_{\mathcal{D}}(\theta_k)$, а современные методы отличаются способом выбора направления, шага и адаптации параметров обучения.

4.16. Проверочные вопросы

1. Почему при фиксированной архитектуре обучение нейронной сети можно рассматривать как задачу оптимизации по вектору параметров?
2. Что такое обучающая выборка и как она задает целевую функцию обучения?
3. Чем функция потерь отличается от функционала ошибки на выборке?
4. Запишите формальную задачу минимизации эмпирического риска.
5. Зачем обучающую выборку делят на train, validation и test?
6. Почему статистики нормализации нужно вычислять только по обучающей подвыборке?
7. Почему задача обучения нейронной сети обычно является невыпуклой?
8. Что означают параметры p_k и η_k в общей формуле итерационного обучения?
9. Как записывается шаг градиентного спуска?
10. Чем отличаются методы нулевого, первого и второго порядка?

11. Почему минимизация ошибки на обучающей выборке не гарантирует хорошего качества на новых данных?
12. В каком случае задача аппроксимации может свестись к решению системы линейных уравнений?

Источники

1. [Robbins H., Monro S. A stochastic approximation method //The annals of mathematical statistics. – 1951. – С. 400-407.](#)

Лекция 5. Общая схема градиентного обучения

Лекция рассматривает градиентное обучение как базовый класс итерационных методов оптимизации параметров нейронной сети. Основные темы: антиградиентное направление, выбор шага, метод наискорейшего спуска и стохастическая оценка градиента.

5.1. Постановка проблемы

После записи обучения как задачи минимизации функции потерь нужно выбрать практический способ двигаться в пространстве параметров. Полный перебор невозможен, поэтому основной вопрос состоит в том, как использовать локальную информацию о производных, чтобы последовательно уменьшать ошибку модели.

5.2. Задача обучения как задача оптимизации

Вспомним постановку задачи обучения для нейронной сети. Пусть нейронная сеть с параметрами w реализует функцию f_w , а обучающая выборка имеет вид

$$\mathcal{D} = \{(x_n, y_n)\}_{n=1}^N. \quad 5.1$$

Для одного объекта задаётся функция потерь

$$L_n(w) = \ell(f_w(x_n), y_n), \quad 5.2$$

а средняя эмпирическая функция потерь на выборке равна

$$L(w) = \frac{1}{N} \sum_{n=1}^N L_n(w). \quad 5.3$$

Тогда обучение сети можно записать как задачу минимизации:

$$w^* \in \arg \min_w L(w). \quad 5.4$$

В простых линейных моделях иногда удаётся получить аналитическое решение. В нейронных сетях функция f_w является композицией большого числа линейных преобразований и нелинейных функций активации, поэтому функция потерь обычно невыпукла, имеет много

параметров и не допускает удобного точного решения. Поэтому обучение почти всегда строится как итерационная оптимизация.

5.3. Общая итерационная схема

Большинство методов обучения можно описать формулой

$$w_{t+1} = w_t + \eta_t p_t, \quad 5.5$$

где:

- w_t - параметры модели на итерации t ;
- p_t - направление изменения параметров;
- $\eta_t > 0$ - шаг обучения;
- t - номер итерации.

Общий алгоритм:

1. Задать начальные параметры w_0 .
2. На очередной итерации вычислить значение функции потерь или её оценку.
3. Найти информацию для выбора направления p_t .
4. Выбрать шаг η_t .
5. Обновить параметры по формуле $w_{t+1} = w_t + \eta_t p_t$.
6. Проверить правило останова.

Разные алгоритмы отличаются тем, какую информацию они используют для выбора p_t , как выбирают η_t и по какой оценке функции потерь делают шаг.

5.4. Градиентный спуск

Градиент функции потерь показывает направление её наибольшего локального роста. Для малого изменения параметров используется линейная аппроксимация

$$L(w + \Delta w) \approx L(w) + \Delta w^T \nabla L(w). \quad 5.6$$

Чтобы уменьшить $L(w)$, естественно двигаться в направлении анти-градиента:

$$p_t = -\nabla L(w_t). \quad 5.7$$

Классический градиентный спуск имеет вид

$$w_{t+1} = w_t + \eta_t p_t = w_t - \eta_t \nabla L(w_t), \quad 5.8$$

где η_t - скаляр, задающий величину шага обучения на итерации $t + 1$.

Если градиент считается по всей обучающей выборке, то

$$\nabla L(w_t) = \frac{1}{N} \sum_{n=1}^N \nabla L_n(w_t). \quad 5.9$$

Такой вариант называют пакетным градиентным спуском (batch gradient descent). Он использует точный градиент эмпирической функции потерь, но одна итерация становится дорогой на больших выборках.

5.5. Метод наискорейшего спуска

Метод наискорейшего спуска также выбирает направление антиградиента:

$$p_t = -\nabla L(w_t). \quad 5.10$$

Отличие состоит в выборе шага. Вместо заранее заданного η_t метод решает одномерную задачу минимизации вдоль найденного направления:

$$\eta_t \in \arg \min_{\eta > 0} L(w_t + \eta p_t). \quad 5.11$$

После этого параметры обновляются так:

$$w_{t+1} = w_t + \eta_t p_t. \quad 5.12$$

Метод хорошо показывает идею локального убывания функции: направление берётся из градиента, а длина шага выбирается так, чтобы вдоль этой прямой получить наибольшее уменьшение. В больших нейронных сетях точный поиск η_t обычно слишком дорог, поэтому на практике чаще используют заранее выбранный шаг или более дешёвые правила его изменения.

Рассмотрим, как метод наискорейшего спуска применяется к простейшей нейросети из одного линейного нейрона в задаче линейной регрессии. Пусть обучающая выборка записана матрицей признаков

$$X = \begin{pmatrix} x_1^\top \\ x_2^\top \\ \vdots \\ x_N^\top \end{pmatrix} \in \mathbb{R}^{N \times d}, \quad y \in \mathbb{R}^N. \quad 5.13$$

Один линейный нейрон без нелинейной активации вычисляет

$$\hat{y} = Xw, \quad w \in \mathbb{R}^d. \quad 5.14$$

В качестве функции потерь возьмём среднеквадратичную ошибку (mean squared error, MSE):

$$L(w) = \frac{1}{2} \|Xw - y\|_2^2. \quad 5.15$$

Обозначим разницу на шаге t через

$$r_t = Xw_t - y. \quad 5.16$$

Так как

$$\frac{1}{2} (Xw_t - y)^\top (Xw_t - y) = \frac{1}{2} (w_t^\top X^\top Xw_t - w_t^\top X^\top y - y^\top Xw_t + y^\top y), \quad 5.17$$

градиент имеет вид

$$\nabla L(w_t) = X^\top Xw_t - X^\top y = X^\top (Xw_t - y) = X^\top r_t, \quad 5.18$$

а направление наискорейшего спуска равно

$$p_t = -X^\top r_t. \quad 5.19$$

Теперь нужно выбрать шаг η_t , минимизирующий ошибку вдоль прямой $w_t + \eta p_t$:

$$\eta_t \in \arg \min_{\eta > 0} \frac{1}{2} \|X(w_t + \eta p_t) - y\|_2^2. \quad 5.20$$

Подставляя $r_t = Xw_t - y$, получаем одномерную квадратичную функцию:

$$\psi(\eta) = \frac{1}{2} \|r_t + \eta Xp_t\|_2^2. \quad 5.21$$

Её производная по η равна

$$\psi'(\eta) = (Xp_t)^\top (r_t + \eta Xp_t). \quad 5.22$$

Условие минимума $\psi'(\eta_t) = 0$ даёт явный оптимальный шаг:

$$\eta_t = -\frac{(Xp_t)^\top r_t}{\|Xp_t\|_2^2}. \quad 5.23$$

Поскольку $p_t = -X^\top r_t$, числитель можно переписать как

$$-(Xp_t)^\top r_t = \|X^\top r_t\|_2^2 = \|\nabla L(w_t)\|_2^2. \quad 5.24$$

Поэтому для линейного нейрона с MSE получаем формулу

$$\eta_t = \frac{\|\nabla L(w_t)\|_2^2}{\|X\nabla L(w_t)\|_2^2}. \quad 5.25$$

После вычисления этого шага параметры обновляются стандартно:

$$w_{t+1} = w_t - \eta_t \nabla L(w_t). \quad 5.26$$

Этот пример показывает, что для квадратичной функции потерь длину шага можно найти аналитически. Однако уже для многослойной сети с нелинейными активациями функция $L(w_t + \eta p_t)$ обычно перестает быть простой квадратичной функцией, и точный поиск шага становится значительно дороже.

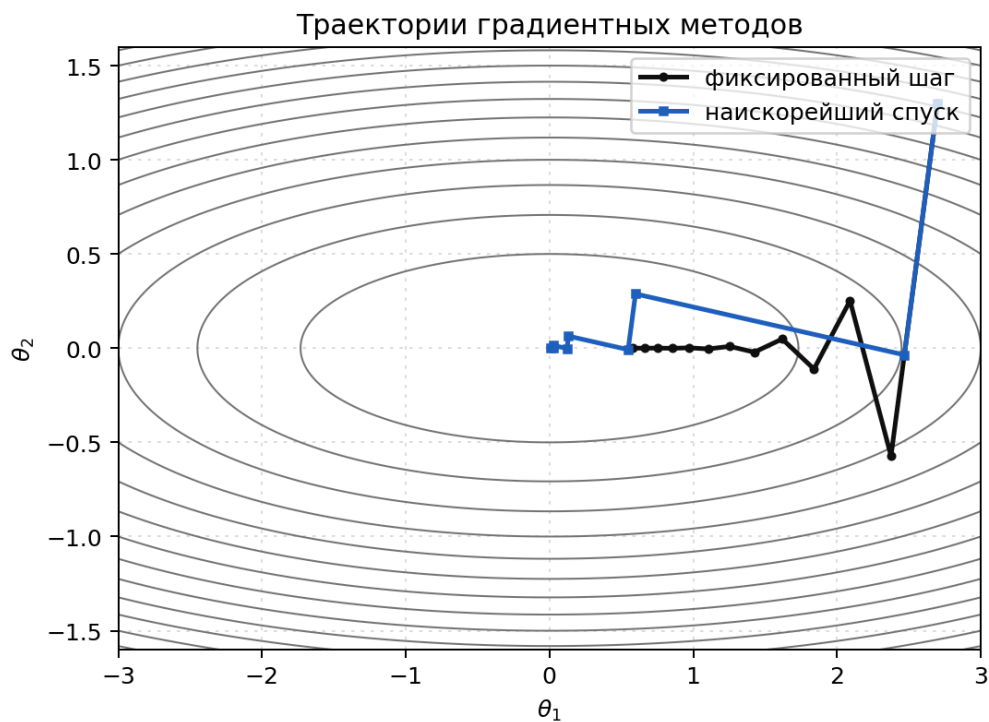


Рис. 5.1. Сравнение траекторий градиентного спуска и метода наискорейшего спуска.

5.6. Поточковый стохастический градиентный спуск

В потоковом стохастическом градиентном спуске (online stochastic gradient descent, online SGD) полный градиент по выборке заменяется градиентом по одному объекту:

$$\nabla L(w_t) \approx \nabla L_{n_t}(w_t). \quad 5.27$$

Обновление имеет вид

$$w_{t+1} = w_t - \eta_t \nabla L_{n_t}(w_t), \quad 5.28$$

где n_t - номер объекта, выбранного на текущей итерации.

На практике объекты часто не выбирают независимо случайно на каждом шаге. Обычно выборку перемешивают в начале эпохи, а затем проходят по объектам последовательно. Один полный проход по всем объектам называют эпохой.

Преимущества потокового SGD:

- одна итерация дешева;
- метод можно применять к потоковым данным;
- шумная оценка градиента иногда помогает не застревать в неглубоких локальных минимумах.

Недостатки:

- направление шага сильно зашумлено;
- траектория обучения может заметно колебаться;
- выбор шага η_t становится особенно важным;
- одиночные объекты плохо используют параллелизм современных графических ускорителей (graphics processing unit, GPU).

5.7. Проблемы базового градиентного алгоритма

Базовый градиентный алгоритм важен как отправная точка, но в обучении нейронных сетей он сталкивается с несколькими типичными трудностями.

1. **Высокая стоимость полного градиента.** Если выборка велика, вычисление $\nabla L(w_t)$ по всем объектам на каждой итерации слишком дорого.
2. **Невыпуклая функция потерь.** У нейронных сетей могут быть локальные минимумы, седловые точки, плато и узкие долины.
3. **Проблемы шага обучения.** Слишком большой шаг приводит к колебаниям или расходимости, слишком малый делает обучение медленным.
4. **Плохое масштабирование координат.** По разным параметрам функция потерь может меняться с разной скоростью, поэтому один общий шаг не всегда удобен.
5. **Инициализация параметров.** Неудачный старт может привести к симметрии нейронов, исчезающим или взрывающимся градиентам.

6. Переобучение. Минимизация обучающей ошибки сама по себе не гарантирует хорошую работу на новых данных.

Эти проблемы объясняют, почему в следующих лекциях появляются более практичные варианты обучения: автоматическое вычисление градиентов, мини-батчи, метод момента (momentum), адаптивные оптимизаторы, регуляризация и специальные правила изменения шага.

5.8. Итог

- Обучение нейронной сети можно рассматривать как задачу минимизации эмпирической функции потерь.
- Общая итерационная схема имеет вид $w_{t+1} = w_t + \eta_t p_t$.
- Градиентный спуск выбирает направление антиградиента.
- Метод наискорейшего спуска дополнительно подбирает шаг одномерной минимизацией.
- Поточковый стохастический градиентный спуск (online SGD) заменяет полный градиент градиентом по одному объекту.
- Базовый градиентный алгоритм прост, но требует развития из-за стоимости вычислений, невыпуклости, выбора шага и переобучения.

5.9. Проверочные вопросы

1. Как записывается задача обучения нейронной сети как задача оптимизации?
2. Что означают w_t , p_t и η_t в общей итерационной схеме?
3. Почему антиградиент является направлением локального убывания функции потерь?
4. Чем метод наискорейшего спуска отличается от обычного градиентного спуска?
5. Как потоковый стохастический градиентный спуск приближает полный градиент?
6. Какие преимущества и недостатки есть у потокового стохастического градиентного спуска?
7. Назовите основные проблемы базового градиентного алгоритма.
8. Почему выбор шага η_t принципиально влияет на сходимость градиентного метода?

Источники

1. Тархов Д. А. Нейронные сети. Модели и алгоритмы. М.: Радиотехника, 2005.
2. Nocedal J., Wright S. J. [Numerical Optimization](#). Springer, 2nd ed., 2006.
3. Robbins H., Monro S. [A stochastic approximation method](#). The Annals of Mathematical Statistics, 1951.
4. [Викиконспекты ИТМО: Обратное распространение ошибки.](#)

Лекция 6. Автоматическое дифференцирование для обучения НС

Лекция объясняет, как эффективно вычислять производные сложных вычислительных графов. Рассматриваются численное, символьное и автоматическое дифференцирование, прямой и обратный режимы, а также метод обратного распространения ошибки для нейронных сетей.

6.1. Постановка проблемы

Градиентные методы требуют производных функции потерь по большому числу параметров. Вручную выводить такие производные для каждой архитектуры невозможно, а численное дифференцирование слишком дорого и неточно. Поэтому нужен общий алгоритм, который использует структуру вычислительного графа и правило цепочки.

6.2. Зачем нужно вычислять градиент

В предыдущей лекции обучение нейронной сети было записано как задача

$$w^* \in \arg \min_w L(w). \quad 6.1$$

Для градиентного обучения нужно вычислять $\nabla L(w)$ быстро и достаточно точно. Это нетривиальная задача: функция потерь нейронной сети является композицией многих операций - линейных слоёв, функций активации, нормализаций и самой функции потерь.

Если градиент вычисляется медленно, обучение становится непрактичным. Если он вычисляется неточно, шаг оптимизации может вести параметры не туда. Поэтому важная часть обучения нейронных сетей - не только выбрать алгоритм обновления весов, но и эффективно получить производные.

6.3. История вопроса

Идея обучать многослойные сети по градиенту долго была связана с проблемой назначения ответственности: если сеть ошиблась на вы-

ходе, нужно понять, какие веса на скрытых слоях внесли вклад в эту ошибку.

Исторически здесь сошлись две линии.

1. В численном анализе развивалось **автоматическое дифференцирование**: способы точно вычислять производные программ, заданных как последовательность элементарных операций.
2. В нейросетях развивался **метод обратного распространения ошибки (backpropagation)**: практическая процедура вычисления градиентов функции потерь по весам многослойной сети.

Впервые метод был описан в 1974 году Александром Галушкиным [1]. Работа Барцева и Охонина (красноярская группа) [2] рассматривала более общий метод («принцип двойственности»), применимый к широкому классу систем. В западной литературе важными историческими источниками считаются диссертация [Werbos](#) о нелинейном анализе чувствительности и работа [Rumelhart, Hinton и Williams](#), которая сделала метод обратного распространения ошибки широко известным в контексте обучения многослойных нейронных сетей.

6.4. Три подхода к вычислению производных

6.4.1. Ручное и символьное дифференцирование

Можно вручную вывести формулу градиента и затем реализовать её в коде. Этот подход даёт точные производные, но плохо масштабируется: формулы быстро разрастаются, а реализация становится источником ошибок.

Символьное дифференцирование автоматизирует вывод формул, но сохраняет ту же проблему: итоговое выражение может быть слишком большим и содержать много повторных вычислений.

Например, для произведения трёх функций

$$F(w) = A(w)B(w)C(w) \tag{6.2}$$

производная равна

$$F'(w) = A'(w)B(w)C(w) + A(w)B'(w)C(w) + A(w)B(w)C'(w). \tag{6.3}$$

Даже в этом простом примере одни и те же промежуточные значения повторяются в разных слагаемых.

Сравнение ручного, символьного, численного и автоматического дифференцирования подробно разобрано в обзоре [Baydin, Pearlmutter, Radul и Siskind](#).

6.4.2. Численное дифференцирование

Производную можно приближать конечными разностями:

$$\frac{\partial L(w)}{\partial w_i} \approx \frac{L(w + \varepsilon e_i) - L(w)}{\varepsilon}, \quad 6.4$$

где e_i - вектор с единицей в i -й координате, а ε - малое число.

Этот способ прост и полезен для проверки градиентов, но плохо подходит для обучения. Если у модели K параметров, нужно примерно $K + 1$ раз вычислить функцию потерь: один раз в точке w и по одному разу для каждого сдвига $w + \varepsilon e_i$.

В практических курсах глубокого обучения (deep learning) конечные разности обычно используют именно как проверку градиента (gradient checking), то есть как тест корректности аналитического или автоматически вычисленного градиента; см., например, материалы [CS231n](#).

6.4.3. Автоматическое дифференцирование

Автоматическое дифференцирование не выводит одну большую символьную формулу. Оно рассматривает вычисление функции как граф простых операций и применяет правило цепочки к этим операциям.

Вычисление функции потерь можно представить как вычислительный граф:

- входные узлы: данные, параметры, гиперпараметры;
- промежуточные узлы: результаты операций;
- выходной узел: скалярная функция потерь L .

Для каждой элементарной операции известна локальная производная. Глобальный градиент получается систематическим применением правила цепочки.

Классическое изложение алгоритмического дифференцирования как техники вычисления производных программ дано в книге [Griewank и Walther](#), а обзор применения автоматического дифференцирования (Automatic Differentiation, AD) в машинном обучении - в статье [Baydin и соавторов](#).

6.5. Правило цепочки и вычислительный граф

Для композиции двух функций

$$L(w) = A(B(w)) \quad 6.5$$

производная равна

$$\frac{dL}{dw} = \frac{dA}{dB} \frac{dB}{dw}. \quad 6.6$$

Для более длинной композиции

$$L(w) = A(B(C(D(w)))) \quad 6.7$$

получаем

$$\frac{dL}{dw} = \frac{dA}{dB} \frac{dB}{dC} \frac{dC}{dD} \frac{dD}{dw}. \quad 6.8$$

Нейронная сеть устроена именно так: выход каждого слоя становится входом следующего слоя, а функция потерь зависит от итогового предсказания. Поэтому граф вычислений естественно описывает и прямой расчёт ответа, и обратный расчёт градиентов.

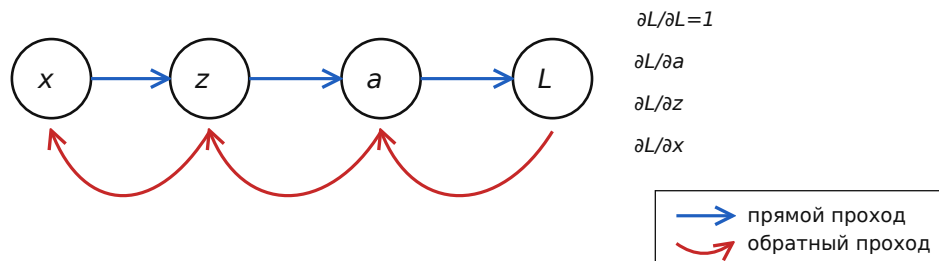


Рис. 6.1. Прямой и обратный проход по вычислительному графу.

6.6. Режим прохода вперёд

В режиме прохода вперёд вместе со значением каждой промежуточной переменной вычисляется её производная по выбранной входной переменной или направлению.

Пусть есть промежуточные переменные

$$u = B(w), \quad v = A(u). \quad 6.9$$

Тогда при проходе вперёд хранятся пары

$$\left(u, \frac{du}{dw}\right), \quad \left(v, \frac{dv}{dw}\right). \quad 6.10$$

Производная v считается сразу:

$$\frac{dv}{dw} = \frac{dv}{du} \frac{du}{dw}. \quad 6.11$$

Прямой режим (forward-mode) удобен, когда входных переменных мало, а выходов много. Но для нейронной сети обычно наоборот: параметров очень много, а функция потерь одна. Если считать производную по каждому параметру отдельно, вычисления становятся дорогими.

Практическое различие между прямым режимом (forward-mode) и обратным режимом (reverse-mode) хорошо видно в документации [JAX](#), где `jacfwd` связывается с прямым режимом, а `jacrev` - с обратным.

6.7. Режим обратного прохода

В режиме прохода назад сначала выполняется обычный прямой проход:

1. вычисляются значения всех промежуточных переменных;
2. эти значения сохраняются;
3. получается скалярная функция потерь L .

Затем начинается обратный проход. Для каждого узла графа вычисляется производная итоговой потери по значению этого узла. Такие величины часто называют сопряжёнными переменными (adjoints).

Если узел u влияет на нескольких следующих узлов $v_1, \dots, v_m$, то

$$\frac{\partial L}{\partial u} = \sum_{j=1}^m \frac{\partial L}{\partial v_j} \frac{\partial v_j}{\partial u}. \quad 6.12$$

Это и есть локальное применение правила цепочки в обратном направлении по графу.

В нейронных сетях обратный режим (backward-mode) называют **методом обратного распространения ошибки**. Слово «ошибка» здесь означает сигнал производной функции потерь, который идёт от выхода сети к более ранним слоям.

Обратный режим автоматического дифференцирования (reverse-mode AD) исторически связан с работами по накоплению сопряжённых

величин; для современных нейросетей стандартная ссылка - статья [Rumelhart, Hinton и Williams](#).

6.8. Пример для одного слоя

Рассмотрим слой

$$z = Wx + b, \quad a = \varphi(z), \quad 6.13$$

где W - матрица весов, b - вектор смещений, φ - функция активации.

Пусть на обратном проходе уже известен градиент по выходу слоя:

$$\delta_a = \frac{\partial L}{\partial a}. \quad 6.14$$

Тогда градиент по преактивации равен

$$\delta_z = \frac{\partial L}{\partial z} = \delta_a \odot \varphi'(z), \quad 6.15$$

где $\odot$ означает покомпонентное умножение.

Градиенты по параметрам слоя:

$$\frac{\partial L}{\partial W} = \delta_z x^\top, \quad \frac{\partial L}{\partial b} = \delta_z. \quad 6.16$$

Градиент, который передаётся на предыдущий слой:

$$\frac{\partial L}{\partial x} = W^\top \delta_z. \quad 6.17$$

Так слой получает входящий обратный сигнал, вычисляет градиенты по своим параметрам и передаёт сигнал дальше назад.

Такая запись слоя соответствует стандартному выводу метода обратного распространения ошибки для сетей прямого распространения (feedforward networks); подробное учебное изложение дано в главе 6 книги [Goodfellow, Bengio и Courville](#).

6.9. Алгоритм обратного распространения ошибки

Метод обратного распространения ошибки удобнее всего понимать как алгоритм над вычислительным графом. Узлы графа хранят промежуточные значения, рёбра показывают зависимости, а обратный проход накапливает производные итоговой потери по значениям узлов.

Для сети из нескольких слоёв алгоритм состоит из следующих шагов.

1. Построить или неявно записать вычислительный граф прямого расчёта.
2. Выполнить прямой проход от входов к функции потерь.
3. Сохранить промежуточные значения, от которых зависят локальные производные.
4. Положить для выходного узла $\partial L / \partial L = 1$.
5. Идти по узлам в обратном топологическом порядке.
6. Для каждого узла u просуммировать вклады всех следующих узлов v из множества $C(u)$, зависящих от u .
7. Если узел является параметром модели, записать найденную величину как компоненту градиента.
8. Передать найденные градиенты оптимизатору, например градиентному спуску или стохастическому градиентному спуску (stochastic gradient descent, SGD).

Основная рекуррентная формула обратного прохода имеет вид

$$\frac{\partial L}{\partial u} = \sum_{v \in C(u)} \frac{\partial L}{\partial v} \frac{\partial v}{\partial u}. \quad 6.18$$

Рассмотрим этот алгоритм на маленьком примере. Пусть функция потерь линейной модели с L_2 -регуляризацией имеет вид

$$L(w) = (w^\top x - y)^2 + \lambda w^\top w. \quad 6.19$$

Введём промежуточные переменные

$$a = w^\top x, \quad b = a - y, \quad c = b^2, \quad d = w^\top w, \quad e = \lambda d, \quad L = c + e. \quad 6.20$$

Такое разложение отделяет сложную формулу от элементарных операций: скалярного произведения, вычитания, возведения в квадрат, умножения на константу и сложения.

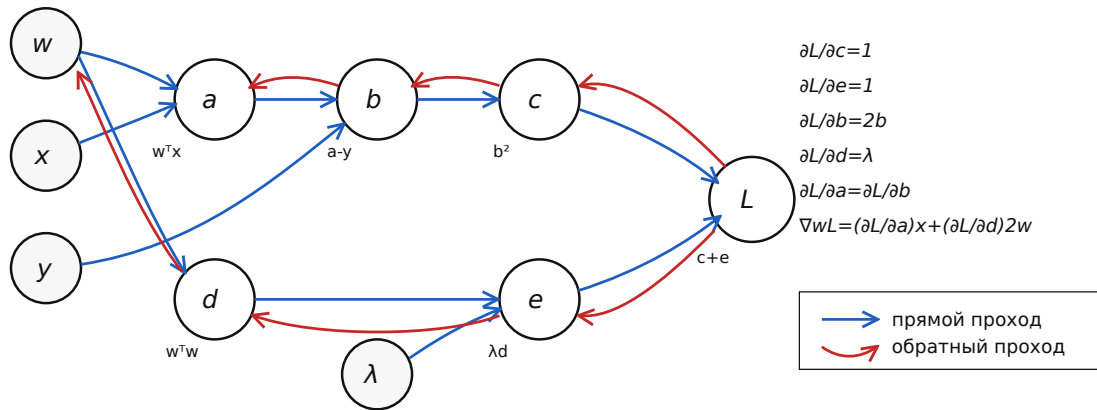


Рис. 6.2. Вычислительный граф для примера обратного распространения ошибки.

Пусть

$$w = (3, 4)^T, \quad x = (1, 2)^T, \quad y = 0, \quad \lambda = 5. \quad 6.21$$

На прямом проходе получаем значения

$$a = 3 \cdot 1 + 4 \cdot 2 = 11, \quad 6.22$$

$$b = a - y = 11, \quad c = b^2 = 121, \quad 6.23$$

$$d = 3^2 + 4^2 = 25, \quad e = \lambda d = 125, \quad 6.24$$

$$L = c + e = 246. \quad 6.25$$

На обратном проходе начинаем с выходного узла:

$$\frac{\partial L}{\partial L} = 1. \quad 6.26$$

Так как $L = c + e$, то

$$\frac{\partial L}{\partial c} = 1, \quad \frac{\partial L}{\partial e} = 1. \quad 6.27$$

Далее

$$\frac{\partial L}{\partial b} = \frac{\partial L}{\partial c} \frac{\partial c}{\partial b} = 1 \cdot 2b = 22, \quad 6.28$$

$$\frac{\partial L}{\partial d} = \frac{\partial L}{\partial e} \frac{\partial e}{\partial d} = 1 \cdot \lambda = 5. \quad 6.29$$

Поскольку $b = a - y$, получаем

$$\frac{\partial L}{\partial a} = \frac{\partial L}{\partial b} \frac{\partial b}{\partial a} = 22 \cdot 1 = 22. \quad 6.30$$

Остаётся собрать вклад в градиент по w . Параметр w влияет на потерю двумя путями: через $a = w^\top x$ и через $d = w^\top w$. Поэтому правило цепочки даёт сумму двух вкладов:

$$\nabla_w L = \frac{\partial L}{\partial a} \nabla_w a + \frac{\partial L}{\partial d} \nabla_w d. \quad 6.31$$

Так как $\nabla_w a = x$ и $\nabla_w d = 2w$, то

$$\nabla_w L = 22(1, 2)^\top + 5 \cdot 2(3, 4)^\top = (22, 44)^\top + (30, 40)^\top = (52, 84)^\top. \quad 6.32$$

Именно это делает обратное распространение ошибки в большой сети: оно не раскрывает всю производную в одну громоздкую формулу, а проходит по графу назад, локально умножает производные и суммирует вклады от всех путей влияния.

Важно различать две части обучения:

- метод обратного распространения ошибки вычисляет градиенты;
- оптимизатор использует эти градиенты, чтобы изменить параметры.

Например, формула

$$w_{t+1} = w_t - \eta_t \nabla L(w_t) \quad 6.33$$

относится к оптимизатору. А метод обратного распространения ошибки отвечает за то, как практически получить $\nabla L(w_t)$ для сложной сети.

Именно такое разделение между вычислением градиента и шагом оптимизации используется в современных фреймворках: например, в PyTorch градиенты вычисляются через [torch.autograd](#), а обновление параметров выполняют оптимизаторы из `torch.optim`.

6.10. Почему режим обратного прохода подходит нейронным сетям

В типичной задаче обучения входов-параметров очень много, а целевой выход один - скалярная функция потерь. В такой ситуации обратный режим позволяет получить градиент по всем параметрам за один прямой и один обратный проход по графу.

Цена этого удобства - память. Для обратного прохода нужно хранить промежуточные значения прямого прохода, иначе локальные производные нельзя будет быстро вычислить. Поэтому современные

фреймворки используют вычислительные графы, кэш промежуточных активаций и специальные техники экономии памяти.

Документация [PyTorch Autograd mechanics](#) описывает эту идею как запись графа операций во время прямого вычисления и последующий обход графа для применения правила цепочки.

6.11. Практический смысл

Автоматическое дифференцирование лежит в основе PyTorch, TensorFlow, JAX и других библиотек глубокого обучения. Пользователь описывает прямое вычисление модели и функции потерь, а система строит граф операций и автоматически вычисляет градиенты.

В PyTorch за это отвечает [torch.autograd](#), в TensorFlow - [tf.GradientTape](#), а в JAX - функциональные преобразования вроде [jax.grad](#).

Это меняет стиль разработки нейросетей:

- исследователь задаёт архитектуру и функцию потерь;
- библиотека отвечает за корректное применение правила цепочки;
- оптимизатор получает готовые градиенты и обновляет параметры.

Поэтому современное обучение нейронных сетей можно представить как связку:

прямой проход → функция потерь → обратный проход → шаг оптимизатора

6.12. Итог

- Градиент нужен для обучения, но ручной вывод производных плохо масштабируется.
- Численное дифференцирование полезно для проверки, но слишком дорого для обучения больших сетей.
- Автоматическое дифференцирование применяет правило цепочки к вычислительному графу.
- Прямой режим (forward-mode) удобен при малом числе входных направлений.
- Обратный режим (backward-mode) эффективен для нейронных сетей, где много параметров и одна скалярная функция потерь.
- Метод обратного распространения ошибки - это обратный режим автоматического дифференцирования, применённый к вычислительному графу нейронной сети.

6.13. Проверочные вопросы

1. Почему для обучения нейронных сетей нужно эффективно вычислять градиент?
2. Чем автоматическое дифференцирование отличается от символического дифференцирования?
3. Почему численное дифференцирование плохо подходит для обучения больших сетей?
4. Что такое вычислительный граф?
5. В чём различие прямого и обратного режимов автоматического дифференцирования?
6. Почему обратный режим особенно удобен для функции потерь нейронной сети?
7. Что вычисляет метод обратного распространения ошибки, а что делает оптимизатор?
8. Запишите формулы обратного прохода для слоя $z = Wx + b$, $a = \varphi(z)$.

Источники

1. Галушкин А. И. Синтез многослойных систем распознавания образов. — М.: «Энергия», 1974.
2. Барцев С. И., Охонин В. А. Адаптивные сети обработки информации. Красноярск : Ин-т физики СО АН СССР, 1986. Препринт N 59Б. — 20 с.
3. Werbos P. J. [Beyond Regression: New Tools for Prediction and Analysis in the Behavioral Sciences](#). PhD thesis, Harvard University, 1974.
4. Rumelhart D. E., Hinton G. E., Williams R. J. [Learning representations by back-propagating errors](#). Nature, 1986.
5. Baydin A. G., Pearlmutter B. A., Radul A. A., Siskind J. M. [Automatic differentiation in machine learning: a survey](#). JMLR, 2018.
6. Griewank A., Walther A. [Evaluating Derivatives: Principles and Techniques of Algorithmic Differentiation](#). SIAM, 2nd ed., 2008.
7. Goodfellow I., Bengio Y., Courville A. [Deep Learning. Chapter 6: Deep Feedforward Networks](#). MIT Press, 2016.
8. [PyTorch: torch.autograd](#).
9. [PyTorch: Autograd mechanics](#).
10. [TensorFlow: Advanced automatic differentiation](#).
11. [JAX: Automatic differentiation](#).
12. [JAX: Forward- and reverse-mode autodiff](#).

13. [CS231n: Neural Networks Part 3, gradient checks.](#)
14. [Автоматическое дифференцирование.](#)

Лекция 7. Практические методы градиентного обучения

Лекция систематизирует практические варианты градиентной оптимизации: стохастические оценки, методы момента, адаптивные шаги по координатам и приближения второго порядка. Материал показывает, как модифицируют базовый градиентный спуск для больших и плохо обусловленных задач.

7.1. Постановка проблемы

Базовый градиентный алгоритм слишком чувствителен к масштабу координат, шуму стохастических градиентов, форме поверхности потерь и выбору шага. Поэтому практическое обучение требует методов, которые сглаживают направление движения, адаптируют шаги, используют мини-батчи и иногда учитывают кривизну функции потерь.

7.2. Проблемы базового градиентного алгоритма

Классический градиентный алгоритм сталкивается с несколькими трудностями.

1. Большая обучающая выборка делает полный градиент дорогим.
2. Невыпуклая функция потерь содержит локальные минимумы, седловые точки и плато.
3. Неудачная инициализация может привести к симметрии нейронов, исчезающим или взрывающимся градиентам.
4. Слишком большой шаг вызывает расходимость, слишком малый - медленное обучение.
5. Направление антиградиента может сильно колебаться в узких долинах функции потерь.
6. Минимизация обучающей ошибки может привести к переобучению.

7.3. Метод наискорейшего спуска

Метод наискорейшего спуска выбирает направление антиградиента:

$$p_t = -\nabla L(w_t), \quad 7.1$$

а шаг η_t подбирает из условия одномерной минимизации вдоль этого направления:

$$\eta_t \in \arg \min_{\eta > 0} L(w_t + \eta p_t). \quad 7.2$$

Обновление:

$$w_{t+1} = w_t + \eta_t p_t. \quad 7.3$$

Метод теоретически естественен, но для больших нейронных сетей полный поиск шага (line search) часто слишком дорог. Поэтому в практическом глубоком обучении чаще используют заранее выбранные или адаптивно изменяемые шаги.

Классическое изложение поиска шага, метода наискорейшего спуска (steepest descent) и методов Ньютона можно найти в книге Nocedal и Wright [Numerical Optimization](#).

7.4. Метод тяжёлого шарика

В практическом обучении чаще используют не полный градиент по всей выборке и не один объект, а мини-батч $B_t \subset \{1, \dots, N\}$. Для него оценка градиента имеет вид

$$g_t = \nabla L_{B_t}(w_t) = \frac{1}{|B_t|} \sum_{n \in B_t} \nabla L_n(w_t). \quad 7.4$$

Если $|B_t| = N$, получается пакетный градиентный спуск; если $|B_t| = 1$, получается потоковый стохастический градиентный спуск. Дальше L_{B_t} обозначает именно мини-батчевую оценку функции потерь.

Метод с моментом добавляет инерцию к обновлениям. Вместо того чтобы каждый раз двигаться только по текущему антиградиенту, алгоритм учитывает предыдущее направление:

$$\Delta w_t = -\eta_t \nabla L_{B_t}(w_t) + \mu \Delta w_{t-1}, \quad 7.5$$

$$w_{t+1} = w_t + \Delta w_t. \quad 7.6$$

Здесь $\mu \in [0, 1)$ - коэффициент момента, часто берут значение около 0.9.

Смысл момента:

- если градиенты несколько шагов направлены примерно одинаково, движение ускоряется;
- если градиенты быстро меняют направление, противоположные компоненты частично компенсируются;
- метод может проходить узкие долины устойчивее, чем обычный SGD.

Если μ слишком велик, инерция начинает опираться на устаревшие направления и может замедлять сходимость или вызывать колебания.

Идея инерционного ускорения восходит к [методу тяжёлого шарика Б. Т. Поляка](#).

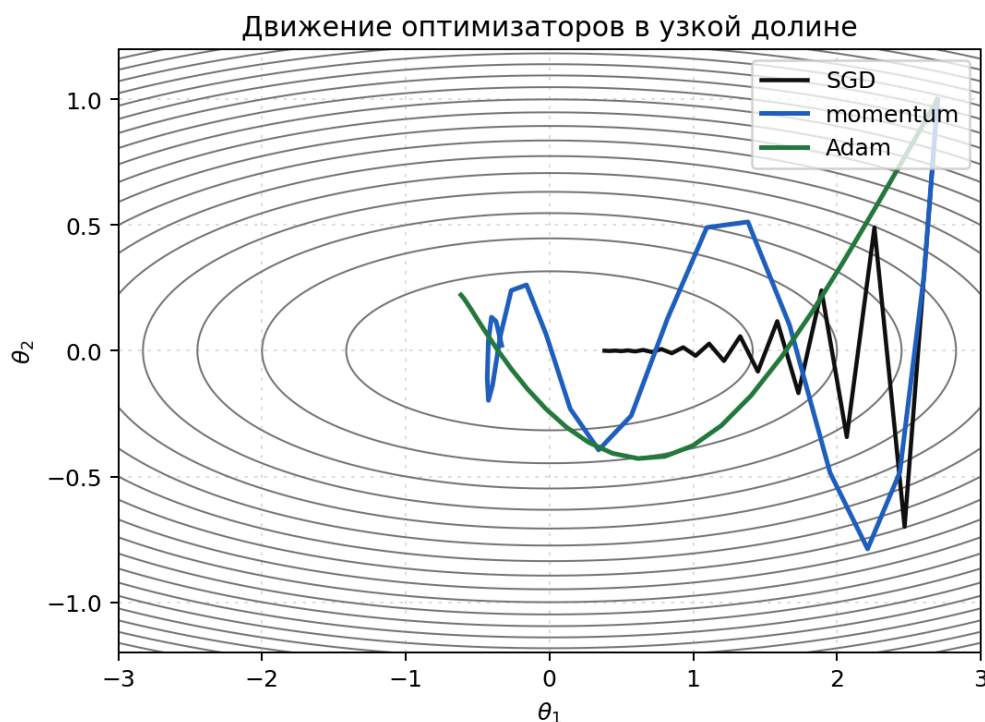


Рис. 7.1. Траектории оптимизаторов в узкой долине функции потерь.

7.5. Метод Нестерова с моментом

В методе Нестерова градиент вычисляется не в текущей точке, а в точке упреждения:

$$\Delta w_t = -\eta_t \nabla L_{B_t}(w_t + \mu \Delta w_{t-1}) + \mu \Delta w_{t-1}, \quad 7.7$$

$$w_{t+1} = w_t + \Delta w_t. \quad 7.8$$

Идея: сначала посмотреть, куда приведет накопленная инерция, и уже там оценить градиент. Это делает шаг более информированным, чем классический метод момента (momentum).

Метод связан с [ускоренными градиентными схемами Ю. Е. Нестерова](#), где упреждающая точка используется для улучшения скорости сходимости в выпуклой постановке.

7.6. Сопряженные градиенты

Методы сопряженных градиентов также строят направление с учетом истории предыдущих направлений. В общем виде:

$$p_t = -\nabla L(w_t) + \beta_t p_{t-1}. \quad 7.9$$

Разные варианты метода отличаются формулой для β_t : Флетчера-Ривса, Полака-Рибьера, Хестенса-Штифеля и другие.

Исходная линейная версия метода сопряженных градиентов была предложена [Hestenes и Stiefel](#); нелинейный вариант [Флетчера-Ривса](#) стал одной из базовых форм метода для безусловной минимизации.

В таблице 7.1 приведены три канонических варианта. Здесь $g_k = -\nabla E(w_k)$ - антиградиент функции ошибки, (a, b) - скалярное произведение, $q_k = g_k - g_{k-1}$.

Таблица 7.1. Формулы различных методов сопряженных градиентов

Метод	Направление поиска	Параметры алгоритма
Метод Флетчера-Ривса	$p_k = g_k + \beta_k p_{k-1}$	$\beta_k = \frac{g_k, g_k}{g_{k-1}, g_{k-1}}$
Метод Полака-Рибьера	$p_k = g_k + \beta_k p_{k-1}$	$\beta_k = \frac{q_k, g_k}{g_{k-1}, g_{k-1}}$
Метод Хестенса-Штифеля	$p_k = g_k + \beta_k p_{k-1}$	$\beta_k = \frac{q_k, g_k}{q_k, p_{k-1}}$

Для квадратичных задач метод сопряженных градиентов имеет сильные теоретические свойства. Для нейронных сетей он менее популярен, чем стохастический градиентный спуск по мини-батчам (mini-batch stochastic gradient descent, mini-batch SGD), метод момента и Adam, потому что стохастические батчи и невыпуклость усложняют устойчивый выбор сопряженных направлений.

7.7. RProp, QuickProp и Delta-delta

В старых алгоритмах обучения нейронных сетей встречаются методы, которые адаптируют шаги по координатам параметров.

RProp использует только знак координат градиента, а не их величину. Для каждого параметра поддерживается собственный шаг. Если знак производной сохраняется, шаг увеличивается; если знак меняется, шаг уменьшается. Это стабилизирует движение там, где величины градиентов плохо масштабированы.

QuickProp аппроксимирует поведение функции ошибки по отдельным координатам и пытается сделать ускоренный шаг, используя информацию о предыдущем изменении градиента.

Delta-delta также изменяет индивидуальные шаги параметров на основе истории изменения градиентов.

Эти методы важны исторически и полезны для понимания идеи покоординатной адаптации, но в современных глубоких сетях чаще используют AdaGrad, RMSprop, Adam и их модификации.

Для исторического контекста см. работы [Riedmiller и Braun](#) по RProp, [Fahlman](#) по QuickProp и [Jacobs](#) по правилу delta-bar-delta.

7.8. AdaGrad

AdaGrad адаптирует шаг обучения отдельно для каждого параметра. Пусть $g_t = \nabla L_{B_t}(w_t)$, а G_t - покоординатная сумма квадратов прошлых градиентов:

$$G_t = G_{t-1} + g_t \odot g_t. \quad 7.10$$

Обновление:

$$w_{t+1} = w_t - \frac{\eta}{\sqrt{G_t} + \varepsilon} \odot g_t. \quad 7.11$$

Здесь $\varepsilon > 0$ предотвращает деление на ноль, а $\odot$ означает покоординатное умножение.

AdaGrad уменьшает шаг для параметров, по которым часто приходят большие градиенты, и хорошо работает с разреженными признаками. Недостаток: накопленная сумма G_t только растет, поэтому шаги могут стать слишком малыми.

Алгоритм AdaGrad введен в работе [Duchi, Hazan и Singer](#) об адаптивных субградиентных методах.

7.9. RMSprop

RMSprop исправляет проблему AdaGrad, заменяя сумму квадратов градиентов экспоненциальным скользящим средним:

$$v_t = \rho v_{t-1} + (1 - \rho) g_t \odot g_t. \quad 7.12$$

Обновление:

$$w_{t+1} = w_t - \frac{\eta}{\sqrt{v_t} + \varepsilon} \odot g_t. \quad 7.13$$

Параметр ρ задает скорость забывания старых градиентов. Благодаря этому RMSprop сохраняет адаптивность шага на протяжении обучения.

RMSprop обычно цитируют по лекции [Tieleman и Hinton Lecture 6.5-rmsprop](#), где предлагается делить градиент на скользящую оценку его недавней величины.

7.10. Adam

Adam объединяет метод момента и адаптацию шага по второму моменту градиента. Для градиента g_t :

$$m_t = \beta_1 m_{t-1} + (1 - \beta_1) g_t, \quad 7.14$$

$$v_t = \beta_2 v_{t-1} + (1 - \beta_2) g_t \odot g_t. \quad 7.15$$

После коррекции смещения:

$$\hat{m}_t = \frac{m_t}{1 - \beta_1^t}, \quad \hat{v}_t = \frac{v_t}{1 - \beta_2^t}. \quad 7.16$$

Обновление:

$$w_{t+1} = w_t - \eta \frac{\hat{m}_t}{\sqrt{\hat{v}_t} + \varepsilon}. \quad 7.17$$

Adam часто является сильным практическим выбором по умолчанию: он устойчив к масштабу градиентов и обычно требует меньше ручной настройки, чем чистый SGD. При этом для некоторых задач SGD с

моментом после хорошей настройки может давать не хуже или лучше обобщение.

Исходное описание Adam дано в работе [Kingma и Ba](#); практические реализации SGD, RMSprop, Adam и L-BFGS также можно сверять с документацией [torch.optim](#).

7.11. Методы второго порядка

Методы второго порядка используют не только значение градиента, но и информацию о кривизне функции потерь. Их цель - учесть локальную геометрию поверхности ошибки и тем самым выбирать более информированное направление движения, чем просто антиградиент.

Для гладкой функции в окрестности точки w_t используют разложение Тейлора до второго порядка:

$$L(w_t + \Delta w) \approx L(w_t) + \Delta w^\top \nabla L(w_t) + \frac{1}{2} \Delta w^\top H_t \Delta w, \quad 7.18$$

где

$$g_t = \nabla L(w_t), \quad H_t = \nabla^2 L(w_t) \quad 7.19$$

- градиент и гессиан функции потерь в текущей точке.

Если считать эту квадратичную аппроксимацию точной локальной моделью, то новое направление ищут как минимум правой части по Δw . Для этого вычисляют производную по Δw и приравнивают ее нулю:

$$\nabla_{\Delta w} \left(L(w_t) + \Delta w^\top g_t + \frac{1}{2} \Delta w^\top H_t \Delta w \right) = g_t + H_t \Delta w = 0. \quad 7.20$$

Отсюда получается линейная система

$$H_t \Delta w = -g_t. \quad 7.21$$

Если H_t обратима, то

$$\Delta w = -H_t^{-1} g_t. \quad 7.22$$

Таким образом, методы второго порядка строят шаг так, чтобы компенсировать не только наклон функции, но и ее локальную кривизну по разным направлениям.

7.11.1. Метод Ньютона

Метод Ньютона задает направление

$$p_t = -H_t^{-1} \nabla L(w_t). \quad 7.23$$

Обновление обычно записывают как

$$w_{t+1} = w_t + \eta_t p_t, \quad 7.24$$

где при идеальном ньютоновском шаге часто берут $\eta_t = 1$, а на практике могут добавлять поиск шага или демпфирование (damping).

Смысл метода Ньютона можно понять так:

- если вдоль некоторого направления кривизна велика, то обратный гессиан уменьшает компоненту шага в этом направлении;
- если кривизна мала, соответствующая компонента шага становится больше;
- при квадратичной функции с положительно определенным гессианом метод сразу учитывает анизотропию поверхности ошибки.

Для квадратичной функции

$$L(w) = \frac{1}{2} w^\top A w - b^\top w + c, \quad 7.25$$

где $A = A^\top \succ 0$, имеем

$$\nabla L(w) = A w - b, \quad \nabla^2 L(w) = A, \quad 7.26$$

поэтому ньютоновский шаг равен

$$\Delta w = -A^{-1}(A w - b) = w^* - w, \quad 7.27$$

где $w^* = A^{-1}b$ - точка минимума. Значит, для точной квадратичной модели метод Ньютона приводит в минимум за один шаг.

Недостатки для больших нейронных сетей:

- гессиан имеет огромный размер;
- обращение матрицы дорого;
- в невыпуклой задаче гессиан может быть не положительно определен;
- стохастические батчи делают оценку кривизны шумной.

Если гессиан не положительно определен, ньютоновское направление может перестать быть направлением убывания. Поэтому в реальных алгоритмах часто используют модификации, которые подменяют H_t^{-1} более устойчивой матрицей.

7.11.2. Общая квазиньютоновская схема

В исходной лекции методы второго порядка описываются через общую формулу

$$p_t = -B_t g_t, \quad 7.28$$

где B_t - матрица, заменяющая обратный гессиан H_t^{-1} .

Тогда обновление имеет вид

$$w_{t+1} = w_t + \eta_t p_t. \quad 7.29$$

Идея квазиньютоновских методов состоит в том, чтобы не обращаться точный гессиан на каждом шаге, а итерационно строить хорошее приближение $B_t \approx H_t^{-1}$. Это позволяет сохранить часть преимуществ ньютоновской схемы и снизить вычислительную цену.

Обычно от матрицы B_t требуют:

- симметричности;
- положительной определенности;
- выполнения секущего условия, связывающего изменение параметров и изменение градиента.

Если обозначить

$$s_t = w_{t+1} - w_t, \quad y_t = g_{t+1} - g_t, \quad 7.30$$

то секущее условие записывают как

$$B_{t+1} y_t = s_t. \quad 7.31$$

Оно означает, что новая аппроксимация обратного гессиана должна правильно воспроизводить хотя бы одно наблюдаемое локальное соотношение между изменением градиента и изменением параметров.

7.11.3. BFGS и L-BFGS

Один из самых известных квазиньютоновских методов - BFGS. В форме для приближения обратного гессиана его обновление можно записать так:

$$\rho_t = \frac{1}{y_t^\top s_t}, \quad 7.32$$

$$B_{t+1} = (I - \rho_t s_t y_t^\top) B_t (I - \rho_t y_t s_t^\top) + \rho_t s_t s_t^\top. \quad 7.33$$

Если выполняется условие кривизны

$$y_t^\top s_t > 0, \quad 7.34$$

то при положительно определенном B_t матрица B_{t+1} также остается положительно определенной. Поэтому направление

$$p_t = -B_t g_t \quad 7.35$$

обычно остается направлением убывания.

Практический смысл BFGS:

- метод накапливает информацию о локальной кривизне из истории шагов;
- не требует явного вычисления гессиана;
- часто работает хорошо на гладких задачах малого и среднего размера.

Для очень больших моделей полная матрица B_t слишком велика, поэтому используют L-BFGS, где хранят только несколько последних пар (s_t, y_t) . Это резко уменьшает память, но всё равно хуже сочетается с шумными мини-батчами, чем методы первого порядка.

Метод L-BFGS восходит к работе [Nocedal](#) о квазиньютоновских матрицах с ограниченной памятью; современное учебное изложение BFGS и L-BFGS также дано в книге [Nocedal и Wright](#).

7.11.4. Структура суммы квадратов и метод Гаусса-Ньютона

Во многих задачах обучения функция потерь имеет вид суммы квадратов невязок:

$$L(w) = \frac{1}{2} \sum_{n=1}^N r_n(w)^2 = \frac{1}{2} \|r(w)\|_2^2, \quad 7.36$$

где $r(w) = (r_1(w), \dots, r_N(w))^T$ - вектор невязок.

Пусть $J(w)$ - якобиан невязок:

$$J(w) = \frac{\partial r(w)}{\partial w}. \quad 7.37$$

Тогда градиент равен

$$\nabla L(w) = J(w)^T r(w), \quad 7.38$$

а гессиан имеет разложение

$$\nabla^2 L(w) = J(w)^T J(w) + \sum_{n=1}^N r_n(w) \nabla^2 r_n(w). \quad 7.39$$

Если невязки уже малы или вторые производные отдельных невязок несущественны, вторым слагаемым часто пренебрегают. Получается приближение

$$H(w) \approx J(w)^T J(w). \quad 7.40$$

Тогда направление Гаусса-Ньютона задается решением системы

$$J(w_t)^\top J(w_t) p_t = -J(w_t)^\top r(w_t). \quad 7.41$$

Это приближение особенно естественно для задач нелинейной наименьшей квадратичной аппроксимации, где исходная функция потерь уже построена как сумма квадратов.

Связь метода Гаусса-Ньютона с задачами нелинейных наименьших квадратов обычно рассматривается вместе с методами Ньютона и Левенберга-Марквардта в курсах [численной оптимизации](#).

7.11.5. Метод Левенберга-Марквардта

Алгоритм Левенберга-Марквардта можно рассматривать как компромисс между градиентным спуском и методом Гаусса-Ньютона.

Направление шага определяется системой

$$(J_t^\top J_t + \lambda_t I) p_t = -J_t^\top r_t, \quad 7.42$$

где

$$J_t = J(w_t), \quad r_t = r(w_t), \quad \lambda_t > 0 \quad 7.43$$

- параметр Левенберга-Марквардта.

После этого параметры обновляют по формуле

$$w_{t+1} = w_t + \eta_t p_t \quad 7.44$$

или, в простом варианте, берут $\eta_t = 1$ и регулируют только λ_t .

Роль параметра λ_t легко понять из двух предельных режимов.

Если λ_t велико по сравнению с собственными значениями $J_t^\top J_t$, то

$$(J_t^\top J_t + \lambda_t I)^{-1} \approx \frac{1}{\lambda_t} I, \quad 7.45$$

и потому

$$p_t \approx -\frac{1}{\lambda_t} J_t^\top r_t = -\frac{1}{\lambda_t} \nabla L(w_t). \quad 7.46$$

Значит, алгоритм ведет себя почти как градиентный спуск с малым шагом.

Если же $\lambda_t \rightarrow 0$, то получаем почти шаг Гаусса-Ньютона:

$$(J_t^\top J_t) p_t \approx -J_t^\top r_t. \quad 7.47$$

Поэтому далеко от решения метод работает осторожно и стабильно, а вблизи решения становится более быстрым и «второпорядковым».

Именно эта адаптивная смена режима делает Левенберга-Марквардта сильным инструментом для малых и средних задач аппроксимации, хотя для больших глубоких сетей его стоимость обычно слишком велика.

Классические источники метода - статьи [Levenberg](#) о нелинейных задачах наименьших квадратов и [Marquardt](#) об оценивании нелинейных параметров.

7.12. Итог

- Классический градиентный спуск считает градиент по всей выборке и дорог на больших данных.
- Поточковый стохастический градиентный спуск (online SGD) использует один объект, но дает шумную оценку градиента.
- Стохастический градиентный спуск по мини-батчам является практическим компромиссом и хорошо использует графические ускорители.
- Метод момента и метод Нестерова с моментом ускоряют движение за счет инерции.
- AdaGrad, RMSprop и Adam адаптируют шаги по координатам параметров.
- Методы второго порядка используют кривизну через гессиан или его приближения.
- Ньютона, BFGS и Левенберга-Марквардта дают более информированные шаги, но обычно хуже масштабируются на большие сети.

7.13. Проверочные вопросы

1. Чем пакетный градиентный спуск, поточковый стохастический градиентный спуск и стохастический градиентный спуск по мини-батчам отличаются по оценке градиента?
2. Почему стохастический градиентный спуск по мини-батчам стал основным вариантом в глубоком обучении?
3. Как метод момента меняет формулу обновления параметров?
4. В чем идея метода Нестерова с моментом?
5. Почему AdaGrad может слишком быстро уменьшить шаг обучения?
6. Как RMSprop решает проблему AdaGrad?
7. Какие два момента оценивает Adam?

8. Почему методы второго порядка редко применяют к большим нейронным сетям напрямую?
9. Как выводится ньютоновский шаг из квадратичной аппроксимации функции потерь?
10. Как квазиньютоновские методы, метод Гаусса-Ньютона и метод Левенберга-Марквардта приближают или стабилизируют второго порядковый шаг?

Источники

1. Bottou L., Curtis F. E., Nocedal J. [Optimization Methods for Large-Scale Machine Learning](#). SIAM Review, 2018.
2. Nocedal J., Wright S. J. [Numerical Optimization](#). Springer, 2nd ed., 2006.
3. [Notes for Numerical Optimization](#).
4. Polyak B. T. [Some methods of speeding up the convergence of iteration methods](#). USSR Computational Mathematics and Mathematical Physics, 1964.
5. Nesterov Yu. E. [A method of solving a convex programming problem with convergence rate \$O\(1/k^2\)\$](#) . Doklady AN SSSR, 1983.
6. Hestenes M. R., Stiefel E. [Methods of Conjugate Gradients for Solving Linear Systems](#). Journal of Research of the National Bureau of Standards, 1952.
7. Fletcher R., Reeves C. M. [Function minimization by conjugate gradients](#). The Computer Journal, 1964.
8. Riedmiller M., Braun H. [A direct adaptive method for faster backpropagation learning: the RPROP algorithm](#). IEEE International Conference on Neural Networks, 1993.
9. Fahlman S. E. [An empirical study of learning speed in back-propagation networks](#). Carnegie Mellon University, 1988.
10. Jacobs R. A. [Increased rates of convergence through learning rate adaptation](#). Neural Networks, 1988.
11. Duchi J., Hazan E., Singer Y. [Adaptive Subgradient Methods for Online Learning and Stochastic Optimization](#). JMLR, 2011.
12. Tieleman T., Hinton G. [Lecture 6.5-rmsprop: Divide the Gradient by a Running Average of Its Recent Magnitude](#). Coursera, 2012.
13. Kingma D. P., Ba J. [Adam: A Method for Stochastic Optimization](#). arXiv, 2014.
14. Nocedal J. [Updating Quasi-Newton Matrices with Limited Storage](#). Mathematics of Computation, 1980.
15. Levenberg K. [A Method for the Solution of Certain Non-Linear Problems in Least Squares](#). Quarterly of Applied Mathematics, 1944.

16. Marquardt D. W. [An Algorithm for Least-Squares Estimation of Nonlinear Parameters](#). Journal of the Society for Industrial and Applied Mathematics, 1963.
17. [Документация PyTorch: torch.optim](#).
18. [Оптимизаторы с постоянным шагом](#).
19. Тархов Д. А. Нейронные сети. Модели и алгоритмы. М.: Радиотехника, 2005.

Лекция 8. Настройка обучения и выбор оптимизатора

Лекция посвящена инженерной стороне обучения нейронных сетей: выбору шага, расписаний шага обучения, инициализации, регуляризации, размера мини-батча и критериев останова. Эти решения связывают формулы оптимизаторов с устойчивым экспериментальным протоколом.

8.1. Постановка проблемы

Даже корректно выбранный оптимизатор может давать плохой результат при неудачном шаге обучения, масштабе инициализации, размере мини-батча или критерии останова. Нужно понимать, какие гиперпараметры определяют ход обучения и как выбирать их так, чтобы модель сходилась, не переобучалась и эффективно использовала вычислительные ресурсы.

8.2. Выбор шага обучения

Шаг обучения η_t - один из главных гиперпараметров оптимизации.

Если η_t слишком велик, то параметры нейросети могут «перескакивать» целевую область, переходить в колебания около искомого минимума или удаляться от целевой функции.

Если η_t слишком мал, то обучение идет слишком медленно, а вычислительные ресурсы расходуются неэффективно.

Для диагностики строят график $L(w_t)$ по итерациям или эпохам. В нормальном режиме среднее функции потерь (далее - потери) в среднем убывают. Рост потерь часто указывает на слишком большой шаг.

8.3. Планы изменения шага обучения

В стохастической оптимизации полезно уменьшать шаг обучения по ходу обучения: в начале нужен достаточно крупный шаг для быстрого движения, в конце - меньший шаг для стабилизации около найденной

области. Функцию изменения величины шага обучения будем называть **планом** обучения.

Пусть t - номер этапа обучения, обычно эпохи или итерации, T - общее число этапов. Перечислим наиболее популярные планы:

Линейный план

$$\eta_t = \eta_0 \left(1 - \frac{t}{T}\right) + \eta_T \frac{t}{T}, \quad \eta_0 > \eta_T > 0. \quad 8.1$$

Полиномиальный план

$$\eta_t = \frac{\eta_0}{(1 + \alpha t)^\beta}, \quad \alpha > 0, \quad \beta \in (0, 1]. \quad 8.2$$

Экспоненциальный план

$$\eta_t = \eta_0 \alpha^t, \quad \alpha \in (0, 1). \quad 8.3$$

Косинусный план

$$\eta_t = \eta_T + (\eta_0 - \eta_T) \frac{1 + \cos(\pi t/T)}{2}. \quad 8.4$$

Параметры $\eta_0, \eta_T, \alpha, \beta$ подбирают по валидационной выборке.

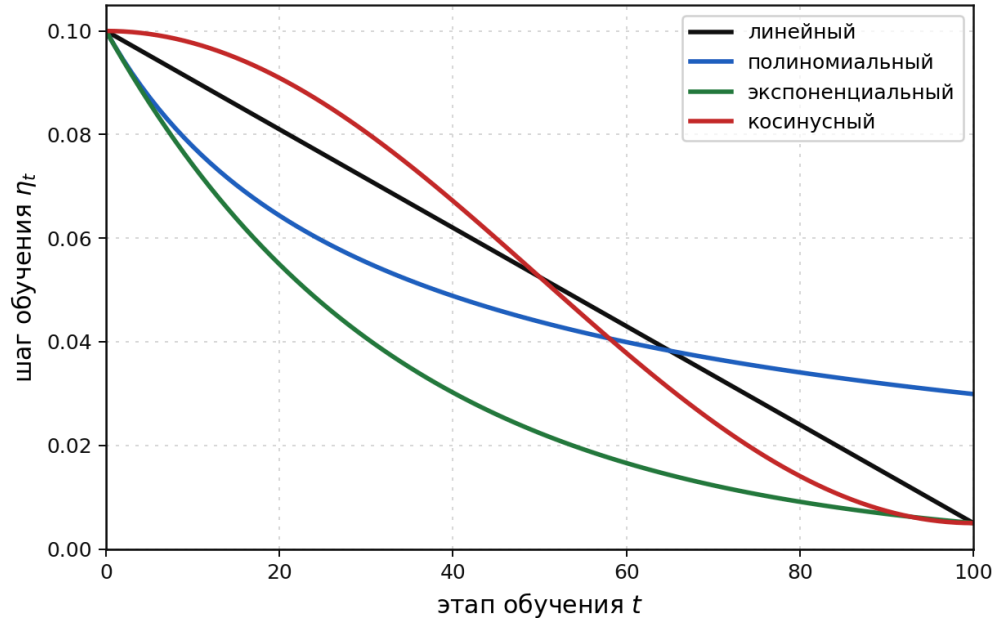


Рис. 8.1. Типовые планы изменения шага обучения.

8.4. Уменьшение шага при плато и разогрев

Уменьшение шага при плато (ReduceLROnPlateau) уменьшает шаг обучения, если последние P итераций или эпох не дают существенного улучшения функции потерь. Обычно используют правило:

- если улучшение меньше ε в течение P блоков;
- уменьшить η в α раз.

Такой план подстраивается под фактическую динамику обучения.

Разогрев (warm-up) - начальное плавное увеличение шага обучения (learning rate). Вместо старта сразу с большим η шаг плавно увеличивают от малого значения до целевого:

$$\eta_t = \eta_{\max} \frac{t}{T_{\text{warmup}}}, \quad 0 \leq t \leq T_{\text{warmup}}. \quad 8.5$$

Разогрев особенно полезен при дообучении предобученных моделей и при обучении больших сетей, где резкий старт с большим шагом может разрушить уже полезные веса.

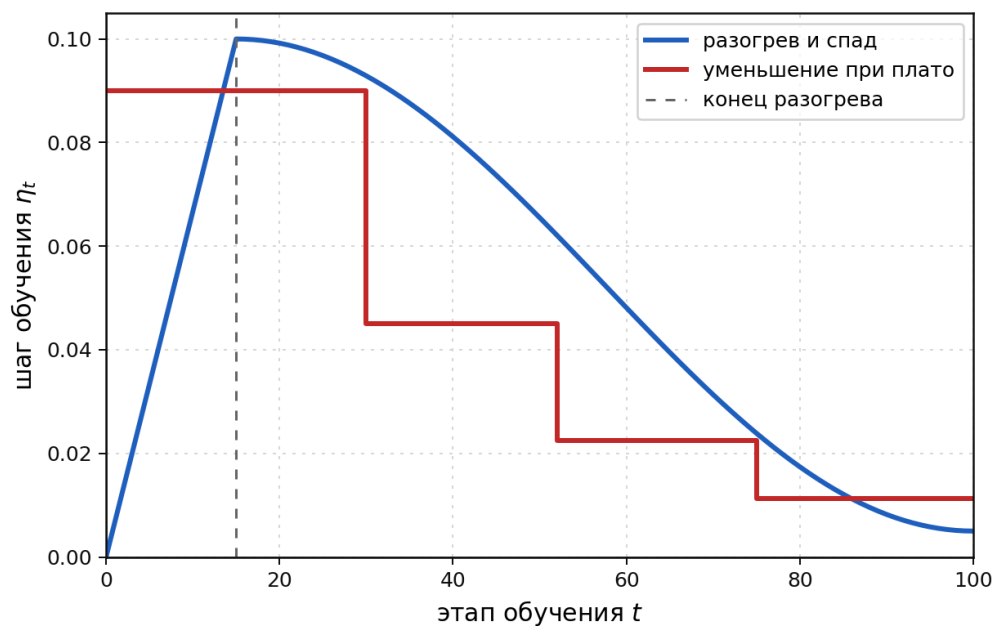


Рис. 8.2. Разогрев и уменьшение шага обучения при плато.

8.5. Инициализация параметров

Инициализация влияет на работу оптимизатора.

Нулевая инициализация весов обычно непригодна: симметричные нейроны получают одинаковые градиенты и обучаются одинаково. Смещения часто можно инициализировать нулем, но веса должны различаться.

Пусть для некоторого слоя:

$$fan_in = n_{in} \quad 8.6$$

- входная ширина слоя (*fan-in*), то есть число входов нейрона, а

$$fan_out = n_{out} \quad 8.7$$

- выходная ширина слоя (*fan-out*), то есть число его выходных связей. Основная идея хорошей инициализации состоит в том, чтобы на первых шагах обучения не допустить слишком быстрого роста или затухания дисперсии сигналов и градиентов при переходе от слоя к слою.

Если веса независимы, имеют нулевое среднее и дисперсию $\text{Var}(w)$, то для линейной комбинации

$$z = \sum_{i=1}^{n_{in}} w_i x_i \quad 8.8$$

при грубом предположении одинаковой дисперсии входов выполняется оценка

$$\text{Var}(z) \approx n_{in} \text{Var}(w) \text{Var}(x). \quad 8.9$$

Поэтому выбор масштаба весов напрямую влияет на то, будут ли активации и градиенты сохранять разумный порядок величины.

8.5.1. Простая случайная инициализация

Простейший вариант - выбирать веса случайно из симметричного распределения с малой дисперсией:

$$w_{ij} \sim U[-a, a] \quad 8.10$$

или

$$w_{ij} \sim \mathcal{N}(0, \sigma^2). \quad 8.11$$

Для равномерного распределения

$$\text{Var}(w_{ij}) = \frac{a^2}{3}, \quad 8.12$$

а для нормального распределения дисперсия сразу равна σ^2 .

Такая инициализация лучше нулевой, но сама по себе не учитывает ширину слоя. Поэтому при большом числе входов слишком большие a или σ ведут к быстрому росту активаций, а слишком малые - к их затуханию.

8.5.2. Xavier/Glorot-инициализация

Инициализация Xavier/Glorot подбирает масштаб весов так, чтобы в среднем сохранять дисперсию сигналов при прямом и обратном проходе. Для нее используют компромисс между fan_in и fan_out :

$$\text{Var}(w) \approx \frac{2}{n_{\text{in}} + n_{\text{out}}}. \quad 8.13$$

В нормальной форме это записывают как

$$w_{ij} \sim \mathcal{N}\left(0, \frac{2}{n_{\text{in}} + n_{\text{out}}}\right). \quad 8.14$$

В равномерной форме границу выбирают из условия $\frac{a^2}{3} = \frac{2}{n_{\text{in}} + n_{\text{out}}}$, то есть

$$w_{ij} \sim U\left[-\sqrt{\frac{6}{n_{\text{in}} + n_{\text{out}}}}, \sqrt{\frac{6}{n_{\text{in}} + n_{\text{out}}}}\right]. \quad 8.15$$

Эта схема особенно хорошо подходит для слоев с сигмоидальной активацией (*sigmoid*) и гиперболическим тангенсом (*tanh*), где слишком крупные веса быстро загоняют нейроны в насыщение.

8.5.3. Не-инициализация

Для активаций типа выпрямленного линейного блока (*rectified linear unit*, ReLU) половина входов после нелинейности часто обнуляется, поэтому полезно брать более крупную дисперсию весов. Инициализация Не использует правило

$$\text{Var}(w) \approx \frac{2}{n_{\text{in}}}. \quad 8.16$$

В нормальной форме:

$$w_{ij} \sim \mathcal{N}\left(0, \frac{2}{n_{\text{in}}}\right). \quad 8.17$$

В равномерной форме:

$$w_{ij} \sim U\left[-\sqrt{\frac{6}{n_{\text{in}}}}, \sqrt{\frac{6}{n_{\text{in}}}}\right]. \quad 8.18$$

Не-инициализация лучше согласована с ReLU, Leaky ReLU и похожими активациями, поскольку помогает сохранить масштаб активаций в глубоких сетях.

8.5.4. Ортогональная инициализация

Ортогональная инициализация строит матрицу весов из случайной матрицы с последующей ортогонализацией. Если Q - ортогональная матрица, то

$$Q^T Q = I \quad 8.19$$

и веса можно задавать как

$$W = \alpha Q, \quad 8.20$$

где $\alpha > 0$ - масштабный коэффициент.

В этом случае сингулярные значения матрицы W равны α , поэтому линейное преобразование меньше искажает нормы векторов, чем произвольная плотная матрица. Это делает ортогональную инициализацию полезной для глубоких линейных слоев и рекуррентных сетей, где важно не разрушать масштаб сигналов при многократном умножении.

8.5.5. Практические замечания

Типичные варианты:

- случайная инициализация из равномерного или нормального распределения полезна как базовая схема, но требует ручного выбора масштаба;
- Xavier/Glorot обычно выбирают для сигмоидальной активации (*sigmoid*) и *tanh*-активаций;
- Не-инициализацию обычно выбирают для ReLU-подобных активаций;
- ортогональная инициализация полезна, когда важно лучше сохранять нормы сигналов и градиентов.

Правильная инициализация помогает избежать исчезающих и взрывающихся градиентов и уменьшает зависимость качества обучения от удачного случайного старта.

8.6. Регуляризация и спад весов

Алгоритм оптимизации минимизирует не только обучающую ошибку, но часто и регуляризованную цель:

$$L_{\text{reg}}(w) = L(w) + \frac{\lambda}{2} \|w\|_2^2. \quad 8.21$$

Здесь λ - коэффициент регуляризации.

Градиент регуляризованной цели:

$$\nabla L_{\text{reg}}(w) = \nabla L(w) + \lambda w. \quad 8.22$$

Соответствующее обновление:

$$w_{t+1} = w_t - \eta_t (\nabla L(w_t) + \lambda w_t). \quad 8.23$$

Спад весов (weight decay) штрафует большие веса, снижает риск переобучения и может делать оптимизацию устойчивее. В современных оптимизаторах важно различать L2-регуляризацию и развязанный спад весов (decoupled weight decay), но базовая идея одна: контролировать сложность модели через параметры.

Для обычного SGD L2-регуляризация и спад весов дают одно и то же обновление. В адаптивных оптимизаторах это уже не так: если добавить λw_t прямо в градиент Adam, регуляризатор тоже пройдет через адаптивное масштабирование координат. В AdamW спад весов отделяют от адаптивного шага:

$$w_{t+1} = (1 - \eta_t \lambda) w_t - \eta_t \frac{\hat{m}_t}{\sqrt{\hat{v}_t} + \varepsilon}. \quad 8.24$$

Поэтому рекомендация «начать с Adam или AdamW» означает: Adam удобен как устойчивый адаптивный оптимизатор, а AdamW предпочтителен, когда нужен явно контролируемый спад весов.

Подробное описание методов регуляризации вынесено в [лекцию 10](#): там отдельно рассматриваются L2-регуляризация, L1-регуляризация, случайное выключение нейронов (dropout) и добавление шума. В лекции 8 регуляризация важна как часть практической настройки обучения: ее силу выбирают вместе с шагом обучения, размером мини-батча и критерием останова.

8.7. Выбор размера мини-батча

На практике чаще используют стохастический градиентный спуск по мини-батчам. Пусть B_t - подмножество объектов выборки размера $|B_t| = B$. Тогда

$$\nabla L(w_t) \approx \frac{1}{B} \sum_{n \in \mathcal{B}_t} \nabla L_n(w_t). \quad 8.25$$

Обновление:

$$w_{t+1} = w_t - \eta_t \frac{1}{B} \sum_{n \in \mathcal{B}_t} \nabla L_n(w_t). \quad 8.26$$

Стохастический градиентный спуск по мини-батчам (mini-batch stochastic gradient descent, mini-batch SGD) балансирует две крайности:

- пакетный градиентный спуск (batch gradient descent, batch GD) использует точный градиент по выборке, но дорог на каждой итерации;
- потоковый стохастический градиентный спуск (online stochastic gradient descent, online SGD) делает дешевые шаги, но градиент слишком шумный;
- стохастический градиентный спуск по мини-батчам использует параллелизм центрального процессора или графического ускорителя (graphics processing unit, GPU) и дает более устойчивую оценку градиента.

Размер мини-батча влияет не только на скорость вычислений, но и на найденное решение: при невыпуклой функции потерь разные размеры батча, инициализации и планы изменения шага могут привести к разным локальным минимумам.

Обзор роли стохастических методов и методов мини-батчей в больших задачах машинного обучения приведен в работе [Bottou, Curtis и Nocedal](#) по оптимизации для крупномасштабного машинного обучения.

Размер мини-батча B задает, сколько объектов используется для одной оценки градиента. Для выборки $\mathcal{D} = \{(\mathbf{x}_i, y_i)\}_{i=1}^N$ и мини-батча $\mathcal{B}_t \subset \{1, \dots, N\}$ оценка градиента имеет вид

$$\hat{\mathbf{g}}_t = \frac{1}{B} \sum_{i \in \mathcal{B}_t} \nabla_w L_i(w_t), \quad B = |\mathcal{B}_t|. \quad 8.27$$

Если объекты выбираются случайно, то такая оценка в среднем совпадает с градиентом эмпирического риска:

$$\mathbb{E}_{\mathcal{B}_t} [\hat{\mathbf{g}}_t] = \nabla L(w_t). \quad 8.28$$

При увеличении B дисперсия оценки обычно уменьшается:

$$\text{Var}(\hat{\mathbf{g}}_t) \approx \frac{1}{B} \text{Var}(\nabla_w L_i(w_t)). \quad 8.29$$

Поэтому большой мини-батч делает шаги стабильнее, но требует больше памяти и может давать менее шумную динамику, иногда ухудшая обобщающую способность. Малый мини-батч дает более шумный градиент, но дешевле по памяти и может помогать выходить из узких областей функции потерь.

Практические ориентиры:

1. Начинать с размера, который эффективно помещается в память графического ускорителя (GPU) или центрального процессора (central processing unit, CPU).
2. При увеличении B проверять, не нужно ли увеличить шаг обучения η .
3. Сравнивать не только скорость одной эпохи, но и качество на валидационной выборке.
4. Для маленьких данных избегать слишком больших батчей, которые делают обучение почти детерминированным и могут усиливать переобучение.

8.8. Выбор критерия останова обучения

Критерий останова определяет, когда обучение следует завершить. Если остановиться слишком рано, модель недообучится; если слишком поздно, возрастает риск переобучения и лишних вычислений.

Базовый критерий - ограничение числа эпох:

$$t \geq T_{\max}. \quad 8.30$$

Более содержательный критерий использует валидационную функцию потерь $L_{\text{val}}(t)$. Обучение останавливают, если за последние P эпох улучшение меньше порога ε :

$$\min_{k=t-P, \dots, t} L_{\text{val}}(k) > L_{\text{val}}^{\text{best}} - \varepsilon. \quad 8.31$$

Такой прием называют ранней остановкой (early stopping). Обычно сохраняют параметры лучшей эпохи:

$$w^* = w_{t^*}, \quad t^* = \arg \min_t L_{\text{val}}(t). \quad 8.32$$

Дополнительные критерии:

1. Норма градиента стала малой:

$$\|\nabla_w L(w_t)\|_2 < \delta. \quad 8.33$$

2. Изменение параметров стало малым:

$$\frac{\|w_{t+1} - w_t\|_2}{\|w_t\|_2 + \varepsilon} < \delta. \quad 8.34$$

3. Целевая метрика на валидационной выборке достигла требуемого уровня.

В практических задачах чаще всего комбинируют ограничение числа эпох, раннюю остановку по валидационной метрике и сохранение лучшей модели.

8.9. Дополнительные метрики при обучении

Функция потерь нужна для оптимизации, но одной ее часто недостаточно для контроля обучения. Поэтому вместе с L_{train} и L_{val} отслеживают дополнительные метрики.

Для классификации часто используют долю правильных ответов (ассурасу):

$$\text{Accuracy} = \frac{1}{N} \sum_{i=1}^N \mathbb{1}[\hat{y}_i = y_i]. \quad 8.35$$

Если классы несбалансированы, полезны точность положительных предсказаний (precision) и полнота (recall):

$$\text{Precision} = \frac{TP}{TP + FP}, \quad \text{Recall} = \frac{TP}{TP + FN}. \quad 8.36$$

Здесь TP - истинно положительные ответы (true positives), FP - ложноположительные ответы (false positives), FN - ложноотрицательные ответы (false negatives), TN - истинно отрицательные ответы (true negatives). При сильном дисбалансе классов ассурасу может быть высокой даже у бесполезной модели, если она почти всегда выбирает частый класс.

Их гармоническое среднее называют F1-мерой (F1 score):

$$F_1 = \frac{2 \text{ Precision Recall}}{\text{Precision} + \text{Recall}}. \quad 8.37$$

Для регрессии обычно отслеживают среднюю абсолютную ошибку (mean absolute error, MAE)

$$\text{MAE} = \frac{1}{N} \sum_{i=1}^N |y_i - \hat{y}_i| \quad 8.38$$

и среднеквадратичную ошибку (mean squared error, MSE)

$$\text{MSE} = \frac{1}{N} \sum_{i=1}^N (y_i - \hat{y}_i)^2. \quad 8.39$$

Диагностические величины помогают понять ход оптимизации:

1. текущий шаг обучения η_t ;
2. норма градиента $\|\nabla_w L(w_t)\|_2$;
3. норма параметров $\|w_t\|_2$;
4. разрыв между обучающей и валидационной ошибкой:

$$\Delta_{\text{gen}}(t) = L_{\text{val}}(t) - L_{\text{train}}(t). \quad 8.40$$

Если L_{train} уменьшается, а L_{val} растёт, это типичный признак переобучения. Если обе ошибки остаются высокими, модель, оптимизатор или признаки могут быть недостаточно выразительными.

Типичная диагностика по кривым обучения:

- L_{train} и L_{val} обе высоки и почти не падают - недообучение или слишком малый шаг;
- L_{train} падает, а L_{val} перестаёт улучшаться - пора уменьшить шаг или остановить обучение;
- L_{train} падает, а L_{val} растёт - переобучение, нужна регуляризация, больше данных или ранняя остановка;
- обе кривые резко растут или колеблются - шаг обучения слишком велик или градиенты нестабильны.

8.10. Как выбирать алгоритм на практике

Практическая схема выбора:

1. Начать с Adam или AdamW на мини-батчах как устойчивого базового варианта.
2. Подобрать шаг обучения по графику потерь и валидационной метрике.
3. Добавить план изменения шага: косинусный, экспоненциальный или уменьшение шага при плато.
4. Для задач, где важна финальная обобщающая способность, попробовать стохастический градиентный спуск (stochastic gradient descent, SGD) с методом момента (momentum) после подбора шага.

5. Следить за переобучением и при необходимости добавить спад весов.
6. Перезапускать обучение с разными инициализациями, если задача невыпуклая и результат нестабилен.

8.11. Итог

- Шаг обучения является одним из главных факторов устойчивости и скорости оптимизации.
- Планы изменения шага и разогрев помогают согласовать быстрый старт обучения с устойчивой сходимостью в конце.
- Инициализация параметров влияет не только на стартовую ошибку, но и на динамику градиентов.
- Регуляризация и спад весов помогают контролировать сложность модели и снижать риск переобучения; подробные методы регуляризации рассмотрены в лекции 10.
- Размер мини-батча управляет компромиссом между шумностью градиента, устойчивостью шага, памятью и скоростью обучения.
- Критерий останова должен опираться не только на число эпох, но и на валидационную динамику.
- Дополнительные метрики помогают отличать успешную оптимизацию от переобучения или недообучения.
- Практический выбор оптимизатора почти всегда связан не только с формулой обновления, но и с настройкой шага, инициализации, регуляризации, размера мини-батча и критерия останова.

8.12. Проверочные вопросы

1. Что происходит при слишком большом и слишком малом шаге обучения?
2. Запишите линейный, полиномиальный, экспоненциальный и косинусный планы изменения шага обучения.
3. В чем идея уменьшения шага при плато?
4. Зачем нужен разогрев?
5. Почему нулевая инициализация весов обычно непригодна?
6. Когда обычно используют Xavier- и He-инициализацию?
7. Как спад весов изменяет функцию потерь и обновление параметров?
8. Как размер мини-батча влияет на дисперсию оценки градиента?

9. Запишите один критерий ранней остановки по валидационной функции потерь.
10. Почему практический выбор оптимизатора нельзя свести только к сравнению формул Adam, RMSprop и SGD?

Источники

1. [Оптимизаторы с постоянным шагом.](#)
2. [Выбор шага обучения.](#)
3. Тархов Д. А. Нейронные сети. Модели и алгоритмы. М.: Радиотехника, 2005.

Лекция 9. Стандартные вычислительные блоки нейросетей

Лекция описывает типовые строительные блоки современных нейронных сетей: линейные, сверточные, рекуррентные и трансформерные слои, нормализацию, нелинейности, эмбединги и функции потерь. Цель — связать математический смысл блока с его ролью в архитектуре.

9.1. Постановка проблемы

Современная архитектура редко проектируется как одна большая формула. Обычно она собирается из повторяемых модулей, у каждого из которых есть своя форма входа, параметры и вычислительный эффект. Чтобы читать и строить архитектуры, нужно понимать эти блоки как математические отображения и знать, когда каждый из них уместен.

9.2. Зачем выделять вычислительные блоки

Современные нейронные сети строятся не как произвольные формулы, а как композиции стандартных преобразований. Каждый такой блок реализует некоторый тип операции над тензором: линейное преобразование, локальную свертку, нормализацию, нелинейность, случайное зануление активаций, рекуррентное обновление состояния и так далее.

В библиотеках вроде `torch.nn` эти блоки оформлены как готовые модули. Для понимания архитектур важнее знать не полный перечень классов, а смысл основных типов преобразований:

- как выглядит формула блока;
- какие данные он обрабатывает;
- в каких задачах его обычно используют.

Эта лекция играет роль карты блоков. Поэтому сложные архитектуры здесь рассматриваются только на уровне входов, выходов и основной формулы. Подробная математика сверточных сетей, рекуррентных ячеек и трансформеров вынесена в лекции 12, 13 и 14 соответственно.

9.3. Линейные слои

Основной линейный блок - полносвязное преобразование `Linear`. Для входного вектора $x \in \mathbb{R}^{d_{\text{in}}}$ оно задается формулой

$$y = Wx + b, \quad 9.1$$

где

$$W \in \mathbb{R}^{d_{\text{out}} \times d_{\text{in}}}, \quad b \in \mathbb{R}^{d_{\text{out}}}. \quad 9.2$$

Если вход батчевый, то одна и та же матрица применяется ко всем объектам батча.

Где применяется:

- в многослойных персептронах;
- в последних классификационных и регрессионных головах;
- в проекциях признаков внутри блоков внимания (attention) и трансформеров (transformers);
- в рекуррентных слоях как часть преобразований входа и состояния.

9.4. Сверточные блоки

9.4.1. Conv1d

Одномерная свертка `Conv1d` обрабатывает последовательности и сигналы. Для входа с каналами $x_c[t]$ выход можно записать как

$$y_k[t] = b_k + \sum_{c=1}^{C_{\text{in}}} \sum_{\tau=0}^{K-1} w_{k,c,\tau} x_c[t + \tau]. \quad 9.3$$

Это локальное линейное преобразование со скользящим окном.

Где применяется:

- в обработке временных рядов;
- в аудиосигналах;
- в биосигналах;
- в моделях для токенов и последовательностей, где полезны локальные шаблоны.

9.4.2. Conv2d

Двумерная свертка `Conv2d` является базовым блоком для изображений. Для одного выходного канала:

$$y_k[i, j] = b_k + \sum_{c=1}^{C_{in}} \sum_{u=0}^{K_h-1} \sum_{v=0}^{K_w-1} w_{k,c,u,v} x_c[i+u, j+v]. \quad 9.4$$

Смысл свертки в том, что один и тот же фильтр применяется ко всем пространственным позициям изображения.

Где применяется:

- в классификации изображений;
- в сегментации;
- в детекции объектов;
- в обработке карт признаков в сверточных нейронных сетях (Convolutional Neural Networks, CNN).

9.4.3. Транспонированная свертка

Транспонированная свертка `ConvTranspose1d` или `ConvTranspose2d` используется для увеличения пространственного или временного разрешения. Формально это линейное преобразование, соответствующее транспонированному оператору обычной свертки.

Если обычная свертка записана как

$$y = Ax, \quad 9.5$$

то транспонированная свертка использует оператор

$$z = A^T y. \quad 9.6$$

Где применяется:

- в декодерах автоэнкодеров;
- в генеративных моделях;
- в задачах повышения разрешения (upsampling);
- в сегментационных архитектурах вида кодировщик-декодировщик (encoder-decoder).

9.5. Субдискретизация и объединение признаков

Блоки объединения признаков (pooling) уменьшают пространственное или временное разрешение и агрегируют локальную информацию.

9.5.1. Объединение признаков с максимизацией

Для окна $R(i, j)$ операция MaxPool2d вычисляет

$$y[i, j] = \max_{(u, v) \in R(i, j)} x[u, v]. \quad 9.7$$

Где применяется:

- в сверточных сетях для уменьшения разрешения;
- для повышения устойчивости к малым сдвигам;
- когда важно сохранить наиболее сильную локальную активацию.

9.5.2. Объединение признаков с усреднением

Для AvgPool2d используется усреднение:

$$y[i, j] = \frac{1}{|R(i, j)|} \sum_{(u, v) \in R(i, j)} x[u, v]. \quad 9.8$$

Отдельно часто используют глобальное среднее объединение признаков (global average pooling), когда усреднение идет по всей пространственной карте признаков.

Где применяется:

- в CNN перед классификационной головой;
- когда нужно сгладить признаки, а не выбирать только максимум;
- в компактных архитектурах без большого числа полносвязных параметров.

9.6. Нелинейные активации

Нелинейность нужна, чтобы сеть не свелась к одной большой линейной функции.

9.6.1. Линейный выпрямитель

Линейный выпрямитель (Rectified Linear Unit, ReLU):

$$\text{ReLU}(x) = \max(0, x). \quad 9.9$$

Где применяется:

- почти во всех современных сетях прямого распространения и сверточных сетях;
- в глубоких архитектурах, где важны простота и стабильность вычислений.

9.6.2. Sigmoid

Функция Sigmoid:

$$\sigma(x) = \frac{1}{1 + e^{-x}}. \quad 9.10$$

Где применяется:

- в бинарной классификации на выходном слое;
- в воротах долгой краткосрочной памяти (Long Short-Term Memory, LSTM) и управляемого рекуррентного блока (Gated Recurrent Unit, GRU);
- в задачах, где нужен выход в диапазоне (0, 1).

9.6.3. Tanh

Функция Tanh:

$$\tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}. \quad 9.11$$

Где применяется:

- в рекуррентных сетях;
- в старых многослойных перцептронах (Multilayer Perceptron, MLP);
- когда полезно иметь выходы с центром около нуля.

9.6.4. Гауссова ошибка линейного блока

Гауссова ошибка линейного блока (Gaussian Error Linear Unit, GELU) обычно записывается как

$$\text{GELU}(x) = x \Phi(x), \quad 9.12$$

где $\Phi(x)$ - функция распределения стандартного нормального закона.

Где применяется:

- в трансформерах;
- в больших языковых моделях;
- в архитектурах, где важна более гладкая нелинейность, чем ReLU.

9.7. Нормализация

Нормализационные блоки стабилизируют распределение активаций и облегчают обучение.

9.7.1. Нормализация по мини-батчу

Для координаты признака по мини-батчу:

$$\mu_B = \frac{1}{m} \sum_{i=1}^m x_i, \quad \sigma_B^2 = \frac{1}{m} \sum_{i=1}^m (x_i - \mu_B)^2. \quad 9.13$$

После этого

$$\hat{x}_i = \frac{x_i - \mu_B}{\sqrt{\sigma_B^2 + \varepsilon}}, \quad y_i = \gamma \hat{x}_i + \beta. \quad 9.14$$

Где применяется:

- в сверточных сетях;
- в глубоких MLP;
- когда обучение идет по достаточно большим мини-батчам (mini-batches).

9.7.2. Нормализация по слою

В нормализации по слою (Layer Normalization, LayerNorm) нормализация выполняется по координатам одного объекта, а не по батчу:

$$\mu = \frac{1}{d} \sum_{j=1}^d x_j, \quad \sigma^2 = \frac{1}{d} \sum_{j=1}^d (x_j - \mu)^2. \quad 9.15$$

$$\hat{x}_j = \frac{x_j - \mu}{\sqrt{\sigma^2 + \varepsilon}}, \quad y_j = \gamma \hat{x}_j + \beta_j. \quad 9.16$$

Где применяется:

- в трансформерах;
- в языковых моделях;
- в задачах, где размер батча мал или нестабилен.

9.8. Рекуррентные блоки

Рекуррентные слои обрабатывают последовательности, поддерживая скрытое состояние.

Формулы ниже нужны, чтобы распознать блок в архитектуре и понять его входы и выходы. Подробно проблемы обучения через время, LSTM, GRU и современные рекуррентные идеи разбираются в лекции 13.

9.8.1. Обычная рекуррентная нейронная сеть

Простейшая рекуррентная нейронная сеть (Recurrent Neural Network, RNN):

$$h_t = \varphi(W_{xh}x_t + W_{hh}h_{t-1} + b). \quad 9.17$$

Где применяется:

- в простых моделях последовательностей;
- в учебных и базовых примерах рекуррентной обработки.

9.8.2. Долгая краткосрочная память

В LSTM есть память ячейки и управляющие ворота:

$$f_t = \sigma(W_f x_t + U_f h_{t-1} + b_f), \quad 9.18$$

$$i_t = \sigma(W_i x_t + U_i h_{t-1} + b_i), \quad 9.19$$

$$\tilde{c}_t = \tanh(W_c x_t + U_c h_{t-1} + b_c), \quad 9.20$$

$$c_t = f_t \odot c_{t-1} + i_t \odot \tilde{c}_t, \quad 9.21$$

$$o_t = \sigma(W_o x_t + U_o h_{t-1} + b_o), \quad h_t = o_t \odot \tanh(c_t). \quad 9.22$$

Где применяется:

- в последовательностях большой длины;
- в обработке текста и временных рядов;
- в задачах, где важно дольше сохранять информацию во времени.

9.8.3. Управляемый рекуррентный блок

GRU проще LSTM и использует два основных механизма ворот:

$$z_t = \sigma(W_z x_t + U_z h_{t-1} + b_z), \quad 9.23$$

$$r_t = \sigma(W_r x_t + U_r h_{t-1} + b_r), \quad 9.24$$

$$\tilde{h}_t = \tanh(W_h x_t + U_h (r_t \odot h_{t-1}) + b_h), \quad 9.25$$

$$h_t = (1 - z_t) \odot h_{t-1} + z_t \odot \tilde{h}_t. \quad 9.26$$

Где применяется:

- в тех же задачах, что и LSTM;
- когда нужна более компактная рекуррентная модель.

9.9. Трансформерные блоки

В разделе `Transformer Layers` PyTorch важнее всего понимать не полный стек классов, а устройство слоя кодировщика (encoder layer).

Здесь трансформерный слой дан как строительный блок. Полный вывод самовнимания через запросы, ключи и значения, причинные маски и различие кодировщика и декодировщика рассматриваются в лекции 14.

9.9.1. Самовнимание

Пусть

$$Q = XW_Q, \quad K = XW_K, \quad V = XW_V. \quad 9.27$$

Тогда масштабированное скалярное произведение внимания (scaled dot-product attention) вычисляется как

$$\text{Attention}(Q, K, V) = \text{softmax} \left(\frac{QK^T}{\sqrt{d_k}} \right) V. \quad 9.28$$

Где применяется:

- в языковых моделях;
- в моделях машинного перевода;
- в обработке последовательностей, где важны дальние зависимости.

9.9.2. Слой кодировщика трансформера

Блок кодировщика (encoder block) сочетает самовнимание (self-attention) и сеть прямого распространения (feedforward network):

$$H' = \text{LayerNorm} (H + \text{SelfAttention} (H)), \quad 9.29$$

$$\text{FFN} (h) = W_2 \phi(W_1 h + b_1) + b_2, \quad 9.30$$

$$H'' = \text{LayerNorm} (H' + \text{FFN} (H')). \quad 9.31$$

Где применяется:

- в кодировщиках трансформеров;
- в BERT-подобных моделях;
- в зрительных трансформерах (vision transformers) и других архитектурах внимания.

9.10. Случайное выключение нейронов

Случайное выключение нейронов (dropout) случайно зануляет часть активаций во время обучения. Если $m_i \sim \text{Bernoulli}(1 - p)$, то при инвертированном случайном выключении нейронов (inverted dropout):

$$y_i = \frac{m_i}{1 - p} x_i. \quad 9.32$$

Во время вывода случайное выключение нейронов отключается, и сеть работает детерминированно.

Где применяется:

- как регуляризация в MLP и CNN;
- в полносвязных головах;
- в трансформерах и рекуррентных моделях как средство борьбы с переобучением.

9.11. Эмбединг

Блок эмбединга (embedding) сопоставляет каждому дискретному индексу обучаемый вектор. Если словарь имеет размер V , а размерность вектора равна d , то матрица эмбедингов

$$E \in \mathbb{R}^{V \times d} \quad 9.33$$

содержит по одной строке на каждый токен. Для индекса i выход равен

$$\text{Embed}(i) = E_{i,:}. \quad 9.34$$

Где применяется:

- в обработке естественного языка (Natural Language Processing, NLP) для слов, субслов и токенов;
- в рекомендательных системах для пользователей и объектов;
- в категориальных признаках табличных моделей.

9.12. Функции потерь

Функция потерь задает, что именно считается ошибкой модели.

9.12.1. MSELoss

Для регрессии:

$$\text{MSE} = \frac{1}{N} \sum_{n=1}^N (\hat{y}_n - y_n)^2. \quad 9.35$$

Где применяется:

- в регрессии;
- в аппроксимации непрерывных функций;
- в задачах восстановления сигнала.

9.12.2. CrossEntropyLoss

Для многоклассовой классификации:

$$\ell(x, y) = -\log \frac{e^{x_y}}{\sum_{c=1}^C e^{x_c}}, \quad 9.36$$

где x_c - логиты классов, а y - правильный класс.

Где применяется:

- в многоклассовой классификации;
- в классификации изображений;
- в языковом моделировании и маркировке токенов.

9.12.3. BCEWithLogitsLoss

Для бинарной классификации:

$$\ell(x, y) = -(y \log \sigma(x) + (1 - y) \log(1 - \sigma(x))), \quad 9.37$$

где $\sigma(x)$ - sigmoid от логита x .

Практически этот блок объединяет sigmoid и бинарную кросс-энтропию в одной численно устойчивой формуле.

Где применяется:

- в бинарной классификации;
- в многометочной классификации (multi-label classification);
- в задачах, где каждый класс предсказывается независимо.

9.13. Как собирают архитектуру из блоков

Реальная нейросеть почти всегда является композицией блоков. Например:

- **MLP:** Linear $\rightarrow$ ReLU $\rightarrow$ Linear $\rightarrow$ ReLU $\rightarrow$ Linear;
- **сверточная нейронная сеть для изображений:** Conv2d $\rightarrow$ BatchNorm $\rightarrow$ ReLU $\rightarrow$ Pooling;
- **LSTM-модель:** Embedding $\rightarrow$ LSTM $\rightarrow$ Linear;
- **кодировщик трансформера:** Embedding $\rightarrow$ Self-Attention $\rightarrow$ LayerNorm $\rightarrow$ FeedForward $\rightarrow$ Dropout.

Поэтому понимание стандартных блоков важнее, чем запоминание полного списка классов библиотеки.

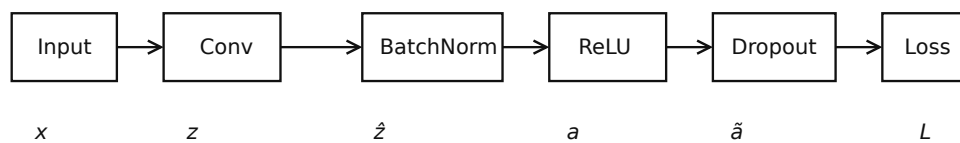


Рис. 9.1. Сборка архитектуры из стандартных вычислительных блоков.

9.14. Итог

- Линейные слои выполняют глобальное аффинное преобразование признаков.
- Свертки извлекают локальные шаблоны и переиспользуют один и тот же фильтр во всех позициях.
- Объединение признаков (pooling) уменьшает разрешение и агрегирует локальную информацию.
- Нелинейные активации делают сеть нелинейной и управляют формой сигнала.
- Нормализация стабилизирует обучение.
- Рекуррентные и трансформерные блоки предназначены для последовательностей и зависимостей во времени или по позиции.
- Случайное выключение нейронов и функции потерь не создают новые признаки, но критически влияют на обучение и обобщение модели.

9.15. Проверочные вопросы

1. Чем линейный слой отличается от свертки?
2. Почему один и тот же сверточный фильтр применяют ко всем позициям входа?

3. В чем разница между максимальным и средним объединением признаков?
4. Почему без нелинейностей глубокая сеть сводится к линейному преобразованию?
5. Когда обычно предпочитают нормализацию по мини-батчу, а когда нормализацию по слою?
6. Зачем в LSTM нужны механизмы ворот?
7. Как самовнимание вычисляет веса влияния одних токенов на другие?
8. Чем случайное выключение нейронов помогает при обучении?
9. Как эмбединг превращает дискретный индекс в вектор признаков?
10. В каких задачах обычно используют MSE, CrossEntropy и BCEWithLogits?

Источники

1. PyTorch 2.12 documentation: [Linear Layers](#).
2. PyTorch 2.12 documentation: [Convolution Layers](#).
3. PyTorch 2.12 documentation: [Pooling Layers](#).
4. PyTorch 2.12 documentation: [Non-linear Activations](#).
5. PyTorch 2.12 documentation: [Normalization Layers](#).
6. PyTorch 2.12 documentation: [Recurrent Layers](#).
7. PyTorch 2.12 documentation: [Transformer Layers](#).
8. PyTorch 2.12 documentation: [Dropout Layers](#).
9. PyTorch 2.12 documentation: [Sparse Layers](#) (`nn.Embedding`).
10. PyTorch 2.12 documentation: [Loss Functions](#).

Лекция 10. Методы регуляризации

Лекция рассматривает регуляризацию как способ ограничить сложность модели и улучшить ее обобщающую способность. Основные методы: L2- и L1-регуляризация, случайное выключение нейронов, добавление шума и их практическое сочетание.

10.1. Постановка проблемы

Минимизация обучающей ошибки сама по себе не гарантирует хорошего качества на новых данных. Если модель слишком гибкая, она может запомнить шум и случайные особенности выборки. Поэтому нужно добавить в обучение ограничения или случайные возмущения, которые делают найденное решение более устойчивым.

10.2. Зачем нужна регуляризация

При обучении нейросети мы минимизируем функцию потерь на конечной обучающей выборке. Если модель слишком гибкая, она может не только уловить полезную зависимость, но и подстроиться под случайный шум в данных. Такое поведение называют переобучением.

Регуляризация - это совокупность приемов, которые ограничивают пространство допустимых решений и тем самым улучшают обобщающую способность модели. Идея состоит не в том, чтобы любой ценой уменьшить обучающую ошибку, а в том, чтобы получить модель, хорошо работающую на новых данных.

С математической точки зрения регуляризацию часто записывают как переход от исходной функции потерь $L(w)$ к регуляризованной цели

$$L_{\text{reg}}(w) = L(w) + \lambda \Omega(w), \quad 10.1$$

где $\Omega(w)$ - штраф за сложность модели, а $\lambda > 0$ - коэффициент регуляризации.

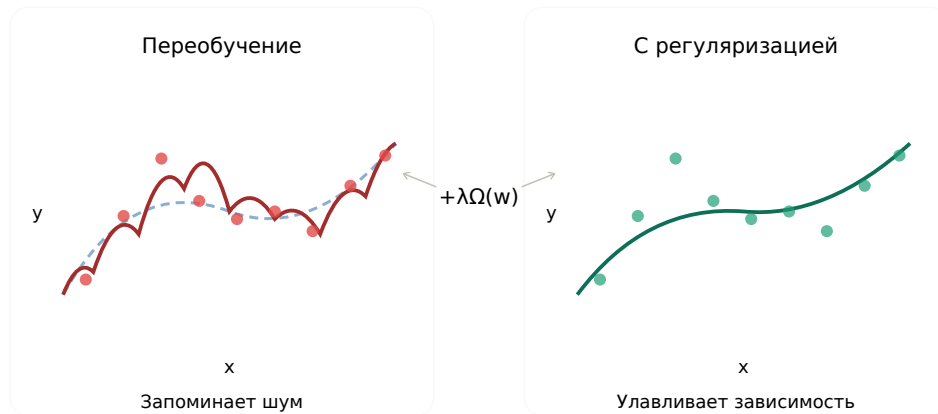


Рис. 10.1. Слева пример воспроизведения функции с выученным шумом, справа - более сбалансированный вариант функции.

Чем больше λ , тем сильнее модель вынуждена предпочитать более простые решения. Если λ выбрать слишком большим, модель недообучится и будет давать слишком грубые прогнозы. Поэтому силу регуляризации обычно подбирают по валидационной выборке.

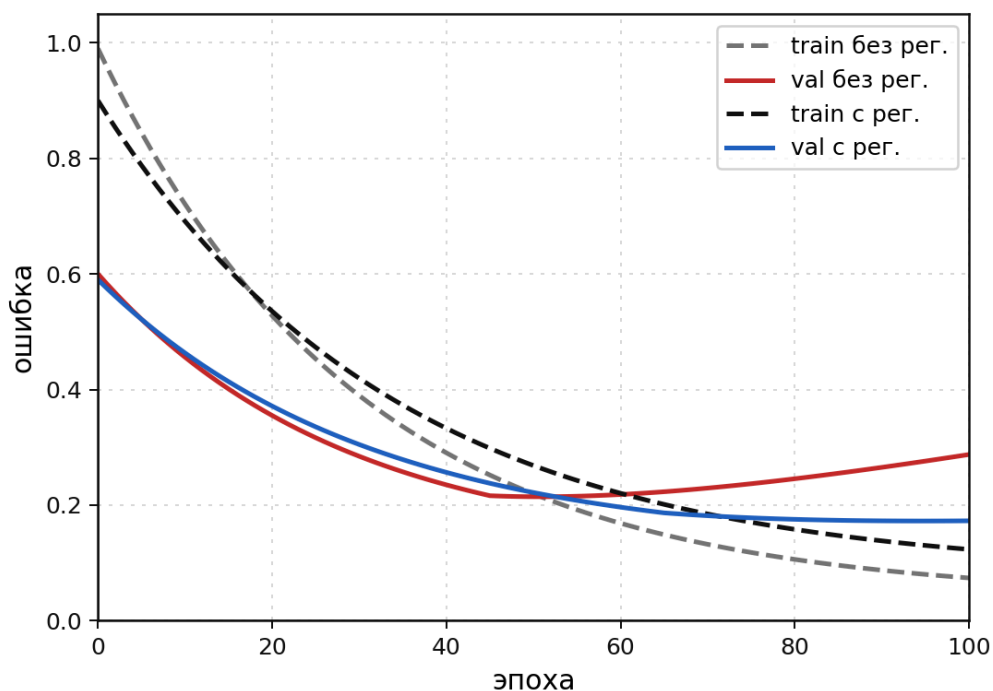


Рис. 10.2. Влияние регуляризации на разрыв между обучающей и валидационной ошибкой.

В этой лекции рассмотрим четыре важных подхода:

- L2-регуляризацию;
- L1-регуляризацию;
- случайное выключение нейронов;
- добавление шума.

10.3. L2-регуляризация

10.3.1. Формула

L2-регуляризация добавляет к функции потерь штраф за сумму квадратов весов:

$$L_{\text{reg}}(w) = L(w) + \frac{\lambda}{2} \sum_{j=1}^m w_j^2. \quad 10.2$$

В векторной форме это же выражение записывают как

$$L_{\text{reg}}(w) = L(w) + \frac{\lambda}{2} \|w\|_2^2. \quad 10.3$$

Часто этот подход называют спадом весов (weight decay). В простом градиентном шаге это проявляется как дополнительное сжатие весов к нулю:

$$w_{t+1} = w_t - \eta \nabla L(w_t) - \eta \lambda w_t = (1 - \eta \lambda) w_t - \eta \nabla L(w_t). \quad 10.4$$

Для обычного стохастического градиентного спуска (stochastic gradient descent, SGD) такая запись эквивалентна L2-регуляризации. Однако в адаптивных оптимизаторах вроде Adam практический вариант спада весов часто реализуют отдельно от градиента регуляризатора, поэтому в современной литературе нередко выделяют специальные схемы вроде AdamW.

Обычно L2-регуляризацию применяют к матрицам весов, но не применяют или применяют слабее к смещениям (bias).

10.3.2. Какой эффект она дает

Штраф на большие веса делает итоговую функцию сети более гладкой. Если веса малы по модулю, то и реакция выходов на изменение входов обычно оказывается менее резкой. Это особенно полезно в ситуациях, когда модель способна построить слишком изломанную зависимость и запомнить случайные особенности обучающей выборки.

L2-регуляризация не зануляет веса полностью в типичном случае, а именно мягко уменьшает их величину. Поэтому она чаще используется как базовый и самый устойчивый способ регуляризации.

10.3.3. Где применяется

- в линейных моделях и многослойных персептронах;

- в сверточных и рекуррентных сетях как стандартная регуляризация весов;
- в больших моделях, где нужно мягко ограничить рост параметров;
- почти во всех сценариях, где есть риск переобучения и нужен простой надежный базовый вариант.

10.4. L1-регуляризация

10.4.1. Формула

L1-регуляризация использует сумму модулей весов:

$$L_{\text{reg}}(w) = L(w) + \lambda \sum_{j=1}^m |w_j| = L(w) + \lambda \|w\|_1. \quad 10.5$$

Для ненулевого веса производная регуляризатора равна

$$\frac{\partial}{\partial w_j} \lambda |w_j| = \lambda \text{sign}(w_j), \quad w_j \neq 0. \quad 10.6$$

Это означает, что L1-регуляризация тянет веса к нулю почти с постоянной силой.

10.4.2. Чем она отличается от L2

У L2-регуляризации производная штрафа равна λw_j , поэтому для очень малых весов давление к нулю тоже становится малым. У L1-регуляризации такого ослабления нет: даже малые веса продолжают активно прижиматься к нулю.

Из-за этого L1-регуляризация чаще приводит к разреженным решениям, когда часть весов становится в точности нулевой. В терминах архитектуры это означает, что некоторые связи фактически перестают участвовать в вычислениях.

10.4.3. Где применяется

- когда нужна разреженность параметров;
- когда важен отбор признаков и желательно автоматически выключать часть связей;
- в компактных моделях, где хотят уменьшить число активных параметров;
- в задачах интерпретируемого машинного обучения как способ выделить небольшой набор важных признаков.

На практике в глубоких сетях L1-регуляризация используется реже, чем L2, потому что она более агрессивна и может заметнее усложнять настройку обучения. Но там, где нужна именно разреженность, она очень полезна.

10.5. Случайное выключение нейронов

10.5.1. Идея метода

Случайное выключение нейронов (dropout) - это стохастическая регуляризация, при которой на каждом шаге обучения часть скрытых нейронов временно выключается. Для каждой активации h_j генерируется случайная маска Бернулли:

$$r_j \sim \text{Bernoulli}(1 - p), \quad 10.7$$

где p - вероятность выключения нейрона. После этого вместо исходной активации используется

$$\tilde{h}_j = r_j h_j. \quad 10.8$$

Без дополнительного масштабирования математическое ожидание уменьшается:

$$\mathbb{E}[\tilde{h}_j] = \mathbb{E}[r_j h_j] = (1 - p)h_j. \quad 10.9$$

В варианте инвертированного случайного выключения нейронов (inverted dropout), который обычно реализуется в современных библиотеках, активные нейроны дополнительно масштабируются:

$$\tilde{h}_j = \frac{r_j}{1 - p} h_j. \quad 10.10$$

Только для этого масштабированного варианта математическое ожидание активации не меняется:

$$\mathbb{E}[\tilde{h}_j] = h_j. \quad 10.11$$

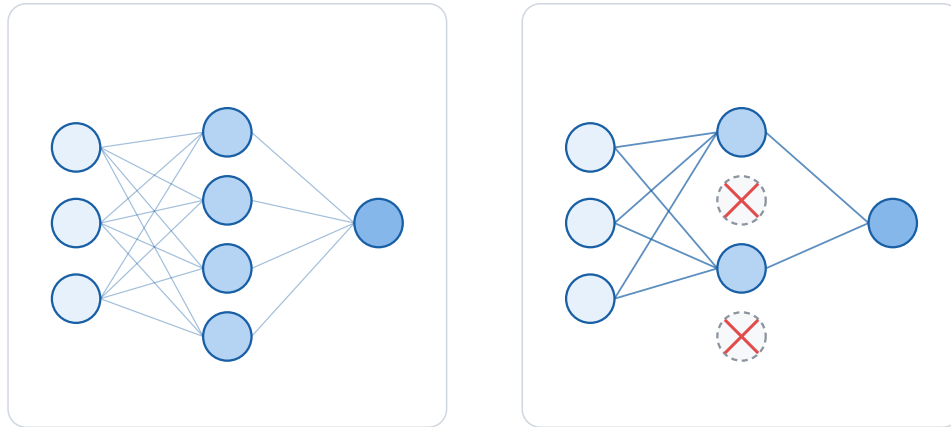


Рис. 10.3. Дропаут - исключение некоторых нейронов из слоя.

10.5.2. Почему случайное выключение нейронов работает

Если сеть слишком сильно полагается на конкретные скрытые нейроны или на их хрупкие комбинации, случайное выключение части узлов делает такую стратегию неустойчивой. Модель вынуждена распределять информацию по нескольким альтернативным путям и учиться более устойчивым представлениям.

С этой точки зрения случайное выключение нейронов можно понимать как обучение ансамбля большого числа похожих подсетей, которые разделяют между собой веса. На каждом шаге SGD обучается немного другая подмодель, а итоговая сеть должна работать устойчиво при множестве таких случайных конфигураций.

10.5.3. Где применяется

- в полносвязных слоях многослойных персептронов;
- в классификационных головах сверточных сетей;
- в некоторых рекуррентных архитектурах и архитектурах трансформеров, где случайное выключение нейронов добавляют к активациям, весам внимания или остаточным ветвям;
- когда модель велика относительно объема обучающих данных и есть явный риск переобучения.

Случайное выключение нейронов обычно применяют только на этапе обучения. На этапе вывода сеть работает детерминированно.

10.6. Добавление шума

10.6.1. Общая идея

Еще один способ регуляризации - намеренно искажать данные или внутренние переменные сети при обучении. Тогда модель учится давать устойчивый ответ не только на точный наблюдаемый вход, но и на его малые возмущения.

Если к входу x добавить случайный шум ε , то вместо исходного объекта используется

$$\tilde{x} = x + \varepsilon. \quad 10.12$$

Часто предполагают, что

$$\varepsilon \sim \mathcal{N}(0, \sigma^2 I). \quad 10.13$$

Аналогично можно зашумлять веса:

$$\tilde{w} = w + \xi, \quad \xi \sim \mathcal{N}(0, \sigma_w^2 I), \quad 10.14$$

или скрытые активации:

$$\tilde{h} = h + \zeta. \quad 10.15$$

10.6.2. Зашумление входов

При зашумлении входов сеть на каждом шаге видит немного разные версии одних и тех же объектов. Это делает итоговую функцию более гладкой и уменьшает чувствительность к малым изменениям признаков.

Такой подход особенно естественен, когда в реальных данных возможны измерительные ошибки, сенсорный шум или неточности ввода. Тогда регуляризация соответствует реальной физике задачи: модель учится быть устойчивой к тем возмущениям, которые действительно могут встретиться на практике.

10.6.3. Зашумление весов и активаций

Шум в весах заставляет модель находить такие области пространства параметров, где небольшое изменение коэффициентов не приводит к резкому ухудшению качества. Это подталкивает оптимизацию к более широким и устойчивым минимумам.

Шум в активациях по смыслу близок к случайному выключению нейронов, но вместо полного зануления нейрона вносит случайное непрерывное возмущение. Оба подхода уменьшают зависимость сети от хрупких внутренних конфигураций.

10.6.4. Где применяется

- при обучении моделей на шумных сигналах и сенсорных данных;
- как способ повысить устойчивость к малым возмущениям входа;
- в шумоподавляющих автоэнкодерах (denoising autoencoders) и похожих архитектурах;
- как часть состязательного и робастного обучения, где модель специально учат быть устойчивой к неблагоприятным возмущениям.

10.7. Сравнение методов

Рассмотренные методы действуют по-разному:

Таблица 10.1. Сравнение методов регуляризации.

Метод	Механизм	Основной эффект	Когда применять
L2-регуляризация	Добавляет штраф за сумму квадратов весов	Мягко уменьшает веса и сглаживает функцию модели	Как базовый устойчивый способ борьбы с переобучением
L1-регуляризация	Добавляет штраф за сумму модулей весов	Поощряет разреженность и может занулять часть связей	Когда нужны отбор признаков, компактность или интерпретируемость
Случайное выключение нейронов	Случайно зануляет часть скрытых активаций при обучении	Ослабляет ко-адаптацию признаков и приближает ансамблирование	В больших сетях, склонных к переобучению скрытых представлений
Добавление шума	Вносит возмущения во входы, веса или активации	Повышает устойчивость к малым изменениям данных и параметров	Для шумных данных, робастного обучения и шумоподавляющих моделей

- L2-регуляризация мягко уменьшает веса и сглаживает модель;
- L1-регуляризация поощряет разреженность и может занулять часть связей;
- случайное выключение нейронов выключает часть узлов и ослабляет коадаптацию признаков;
- добавление шума делает модель устойчивее к возмущениям данных, весов или активаций.

На практике методы можно комбинировать. Например, часто вместе используют спад весов и случайное выключение нейронов, а для данных дополнительно применяют аугментации и шум. Однако слишком сильная комбинация нескольких видов регуляризации может привести к недообучению, поэтому итоговый набор приемов всегда проверяют по валидационной выборке.

10.8. Итог

Регуляризация нужна для того, чтобы сеть не просто запоминала обучающие данные, а выделяла устойчивую зависимость, сохраняющуюся на новых примерах. L2-регуляризация уменьшает величину весов, L1-регуляризация делает решение более разреженным, случайное выключение нейронов обучает сеть быть устойчивой к выпадению части внутренних признаков, а добавление шума повышает устойчивость модели к возмущениям.

Выбор метода зависит от типа модели и цели обучения. Если нужен простой и надежный способ борьбы с переобучением, обычно начинают с L2-регуляризации. Если важна разреженность, добавляют L1. Если модель велика и склонна к коадаптации скрытых признаков, используют случайное выключение нейронов. Если задача предполагает шумные данные или нужна повышенная устойчивость, полезно добавление шума.

10.9. Проверочные вопросы

1. Почему минимизация только обучающей ошибки может приводить к переобучению?
2. Как записывается регуляризованная функция потерь в общем виде?
3. В чем различие между L1- и L2-регуляризацией с точки зрения поведения весов?

4. Почему L1-регуляризация чаще приводит к разреженным решениям?
5. Как случайное выключение нейронов изменяет вычисления скрытых активаций на этапе обучения?
6. Почему случайное выключение нейронов можно интерпретировать как приближение к ансамблированию?
7. Чем зашумление входов отличается по смыслу от L2-регуляризации?
8. Почему слишком сильная регуляризация может ухудшить качество модели?

Источники

1. Deep Machine Learning: [Regularization intro](#).
2. Deep Machine Learning: [Weights regularization](#).
3. Dive into Deep Learning: [Weight Decay](#).
4. Simon J. D. Prince. *Understanding Deep Learning*. Chapter 9: Regularization. <https://udlbook.github.io/udlbook/>.

Лекция 11. Обучение без учителя

Лекция вводит обучение без учителя как постановку, где модель извлекает структуру данных без явных целевых меток. Основное внимание уделяется латентным переменным, генеративным моделям, правдоподобию и метрикам качества синтеза.

11.1. Постановка проблемы

Во многих задачах доступны большие наборы объектов без разметки. Нужно научиться находить в них скрытую структуру, компактные представления, кластеры или вероятностное распределение, из которого можно порождать новые примеры. Эта постановка требует других критериев качества, чем обычное предсказание метки.

11.2. Что такое обучение без учителя

В обучении с учителем мы работали с парами вида (x_i, y_i) и строили отображение от входа к целевой метке или числовому ответу. В обучении без учителя доступны только наблюдаемые данные

$$\{x_i\}_{i=1}^I, \quad 11.1$$

а явных меток нет.

Это означает, что модель должна извлекать структуру из самих данных. В зависимости от цели она может:

- выявлять скрытую организацию набора данных;
- делить объекты на кластеры;
- находить выбросы;
- сжимать данные;
- восстанавливать или очищать испорченные примеры;
- генерировать новые объекты, похожие на обучающие.

Таким образом, обучение без учителя - это не один алгоритм, а целый класс задач, объединенных отсутствием разметки.

11.3. Скрытые переменные как внутренняя структура данных

Во многих моделях обучения без учителя (unsupervised models) вводят скрытую, или латентную, переменную

$$z, \quad 11.2$$

которая должна описывать существенные свойства объекта x в более компактной форме. Обычно размерность латентного описания меньше размерности исходных данных, поэтому z можно понимать как сжатое представление объекта.

Идея состоит в том, что наблюдаемый объект может быть сложным, но его вариативность часто определяется меньшим числом внутренних факторов:

- для изображения лица это могут быть поза, освещение, выражение;
- для аудиосигнала - тембр, высота, шум;
- для текста - тема, стиль, длина, синтаксические особенности.

Латентная переменная не наблюдается напрямую, но модель пытается либо вычислить ее по данным, либо, наоборот, построить данные по ней.

11.4. Таксономия моделей обучения без учителя

11.4.1. От данных к скрытой переменной

Один класс моделей строит отображение

$$x \mapsto z. \quad 11.3$$

В этом случае модель получает объект и переводит его во внутреннее представление. Простейший пример - кластеризация k -means, где каждому объекту сопоставляется номер кластера

$$z \in \{1, 2, \dots, K\}. \quad 11.4$$

Такие модели полезны, когда нас интересует структура данных: группы похожих объектов, компактные представления или признаки для дальнейшего анализа.

11.4.2. От скрытой переменной к данным

Другой класс моделей строит отображение

$$z \mapsto x. \quad 11.5$$

Если на пространстве латентных переменных задано распределение

$$p(z), \quad 11.6$$

то можно:

1. сэмплировать латентную переменную $z \sim p(z)$;
2. отобразить ее в пространство данных;
3. получить новый пример x^* .

Такие модели называют **генеративными**, потому что они умеют порождать новые данные со статистикой, похожей на статистику обучающей выборки.

11.4.3. Генеративные и вероятностные модели

Не всякая генеративная модель является вероятностной. Однако важный подкласс генеративных моделей не только синтезирует новые объекты, но и задает вероятность наблюдаемого примера:

$$p(x \mid \phi), \quad 11.7$$

где ϕ - параметры модели.

Такие модели называют **вероятностными генеративными моделями**. Они особенно ценны, потому что позволяют:

- генерировать новые примеры;
- количественно сравнивать модели по вероятности тестовых данных;
- оценивать, насколько объект типичен для данного распределения;
- обнаруживать аномалии и выбросы.

К этому классу относятся, в частности:

- вариационные автоэнкодеры;
- нормализующие потоки (normalizing flows);
- диффузионные модели.

Генеративно-сопоставительная сеть (Generative Adversarial Network, GAN) тоже является генеративной моделью, но обычно не задает явную вероятность $p(x \mid \phi)$.

11.4.4. Снижение размерности и главные компоненты

Не все задачи обучения без учителя являются генеративными. В снижении размерности цель состоит в том, чтобы заменить исходный объект $x \in \mathbb{R}^D$ более компактным представлением $z \in \mathbb{R}^d$, где $d < D$, и при этом сохранить главную изменчивость данных.

В методе главных компонент (Principal Component Analysis, PCA) ищут ортонормированную матрицу направлений

$$U_d = [u_1, \dots, u_d] \in \mathbb{R}^{D \times d}, \quad U_d^\top U_d = I, \quad 11.8$$

и строят код

$$z = U_d^\top (x - \mu), \quad 11.9$$

где μ - средний объект выборки. Восстановление имеет вид

$$\hat{x} = \mu + U_d z. \quad 11.10$$

PCA выбирает направления максимальной дисперсии и дает линейное латентное пространство. Это важная базовая модель: она показывает, что обучение без учителя может решать задачу представления данных даже без генерации новых объектов.

11.4.5. Автоэнкодеры и обнаружение аномалий

Автоэнкодер строит две функции:

$$z = e_\phi(x), \quad \hat{x} = g_\psi(z), \quad 11.11$$

где e_ϕ - кодировщик (encoder), а g_ψ - декодировщик (decoder). Параметры подбирают по ошибке восстановления:

$$L(\phi, \psi) = \sum_{i=1}^I \left\| x_i - g_\psi(e_\phi(x_i)) \right\|_2^2. \quad 11.12$$

Если размерность или структура z ограничены, модель вынуждена сохранять наиболее важные факторы изменения данных. Поэтому автоэнкодеры используют для сжатия, очистки сигналов, предварительного обучения и построения признаков.

Та же идея дает простой способ обнаружения аномалий. Если модель обучалась на типичных объектах, то для необычного объекта ошибка восстановления

$$r(x) = \left\| x - g_\psi(e_\phi(x)) \right\|_2 \quad 11.13$$

может стать большой. Тогда объект считают выбросом, если $r(x) > \tau$, где τ выбирают по валидационной выборке или допустимой частоте ложных срабатываний.



Рис. 11.1. Автоэнкодер сжимает объект в латентный код и восстанавливает его по ошибке реконструкции.

11.5. Как обучают вероятностные генеративные модели

Если модель задает вероятность наблюдаемого объекта, то естественно подбирать параметры так, чтобы обучающие примеры имели как можно большую вероятность. Тогда функция потерь записывается как сумма отрицательных логарифмов правдоподобия:

$$L(\phi) = - \sum_{i=1}^I \log p(x_i | \phi). \quad 11.14$$

Минимизация этой функции эквивалентна максимизации правдоподобия обучающей выборки.

Такая постановка важна по двум причинам.

Во-первых, она дает формальный критерий обучения. Во-вторых, условие нормировки вероятностей означает, что повышение вероятности реальных примеров автоматически уменьшает вероятность точек, далеких от обучающего распределения. Поэтому вероятностная модель не просто воспроизводит отдельные образцы, а приближается ко всей структуре данных.

11.6. Какая генеративная модель считается хорошей

В главе 14 выделяются несколько желательных свойств генеративной модели.

11.6.1. Эффективная генерация

Модель должна быстро строить новые примеры и по возможности хорошо использовать параллелизм современного оборудования.

11.6.2. Высокое качество примеров

Сгенерированные объекты должны быть правдоподобны и трудно отличимы от реальных данных.

11.6.3. Полнота покрытия распределения

Недостаточно научиться порождать только красивый, но узкий поднабор данных. Хорошая модель должна покрывать всю обучающую вариативность. Иначе возникает эффект частичного покрытия распределения, когда модель умеет воспроизводить лишь некоторые типы объектов.

11.6.4. Хорошо устроенное латентное пространство

Каждому латентному коду z должен соответствовать реалистичный объект x , а малые изменения в z должны вызывать плавные и осмысленные изменения в данных. Это особенно важно для интерполяции, редактирования и управления генерацией.

11.6.5. Разделимость факторов в латентном коде

Желательно, чтобы отдельные координаты или направления в латентном пространстве отвечали за интерпретируемые свойства данных. Тогда изменение одной компоненты может, например, менять освещение на изображении, стиль текста или темп аудио, не разрушая остальные свойства.

11.6.6. Эффективное вычисление правдоподобия

Если модель вероятностная, полезно уметь быстро и точно вычислять

$$p(x \mid \phi)$$

11.15

для новых объектов. Но на практике разные классы моделей дают разные компромиссы: одни хорошо сэмпляют, но не умеют точно считать правдоподобие (likelihood), другие, наоборот, удобны для оценки плотности, но менее удобны для генерации.

11.7. Почему не существует одной идеальной модели

Практика показывает, что современные генеративные архитектуры решают задачу по-разному:

- GAN обычно дает очень качественные примеры и быстрый сэмплинг, но страдает от неполного покрытия распределения;
- вариационный автоэнкодер (Variational Autoencoder, VAE) удобен из-за латентного пространства, но может генерировать менее резкие данные;
- нормализующий поток позволяет точно считать правдоподобие, но накладывает ограничения на архитектуру преобразований;
- диффузионные модели часто дают очень качественные результаты, но генерация у них сравнительно медленная.

Поэтому выбор модели определяется целью:

- если важнее реалистичность примеров, выбирают один класс методов;
- если важно оценивать вероятность и искать выбросы, выбирают другой;
- если нужен управляемый латентный код, смотрят на модели с хорошо организованным пространством представлений.

11.8. Как измерять качество генеративных моделей

11.8.1. Правдоподобие на тестовой выборке

Для вероятностной модели естественно сравнивать значение правдоподобия на тестовых данных, а не на обучающих. Если измерять только обучающую выборку, модель может просто запомнить отдельные точки и дать между ними очень малую вероятность. Тогда обучающее правдоподобие будет большим, но обобщение окажется плохим.

Тестовое правдоподобие отражает сразу два аспекта:

- насколько модель действительно уловила структуру данных;

- насколько полно она покрывает распределение, а не только часть примеров.

Однако эта метрика применима не ко всем моделям:

- у GAN нет явной функции вероятности;
- для VAE и диффузионных моделей правдоподобие оценивать труднее;
- у нормализующего потока правдоподобие можно вычислять точно и эффективно.

11.8.2. Оценка Inception

Для изображений часто применяют оценку Inception (Inception Score, IS). Она опирается на заранее обученный классификатор. Идея у метрики двойная:

- каждое сгенерированное изображение должно уверенно относиться к одному содержательному классу;
- совокупность всех сгенерированных изображений должна покрывать разные классы, а не только один.

Формально метрика использует среднюю дивергенцию Кульбака-Лейблера (Kullback-Leibler divergence, KL-дивергенция) между условным распределением классов для конкретного сэмпла и средним распределением классов по всем сгенерированным данным:

$$IS = \exp \left(\frac{1}{I} \sum_{i=1}^I D_{KL}(p(y | x_i^*) \parallel p(y)) \right), \quad 11.16$$

где

$$p(y) = \frac{1}{I} \sum_{i=1}^I p(y | x_i^*). \quad 11.17$$

Чем лучше изображение похоже на объект одного класса и чем равномернее покрываются классы в целом, тем выше значение IS.

Ограничения метрики:

- она по сути рассчитана на ImageNet-подобные изображения;
- результат зависит от выбранного классификатора;
- она почти не награждает внутриклассовое разнообразие.

11.8.3. Расстояние Фреше-Inception

Расстояние Фреше-Inception (Fréchet Inception Distance, FID) тоже ориентировано на изображения, но сравнивает уже не отдельные мет-

ки классов, а распределения признаков реальных и сгенерированных изображений в глубоком признаковом пространстве.

Если признаки реальных изображений аппроксимированы нормальным распределением с параметрами (μ_r, Σ_r) , а признаки сгенерированных изображений - распределением (μ_g, Σ_g) , то

$$\text{FID} = \left\| \mu_r - \mu_g \right\|_2^2 + \text{Tr} \left(\Sigma_r + \Sigma_g - 2(\Sigma_r \Sigma_g)^{\frac{1}{2}} \right). \quad 11.18$$

Интуитивно:

- если сгенерированные примеры похожи на реальные по смысловой структуре, FID уменьшается;
- если им не хватает реалистичности или разнообразия, FID растет.

Эта метрика чувствительна и к качеству, и к разнообразию, но зависит от того, какую информацию сохраняет используемая сеть признаков.

11.8.4. Точность и полнота для многообразий

Чтобы отдельно оценивать реалистичность и покрытие, используют идею пересечения двух многообразий:

- **многообразие данных (data manifold)** - область пространства, где лежат реальные примеры;
- **многообразие модели (model manifold)** - область, где лежат примеры, синтезируемые моделью.

Тогда:

- **точность (precision)** - доля сгенерированных примеров, попадающих в многообразие реальных данных;
- **полнота (recall)** - доля реальных примеров, покрываемых многообразием модели.

Если точность высока, модель чаще генерирует правдоподобные объекты. Если полнота высока, модель покрывает большую часть реального распределения и не сужает его до небольшого подмножества.

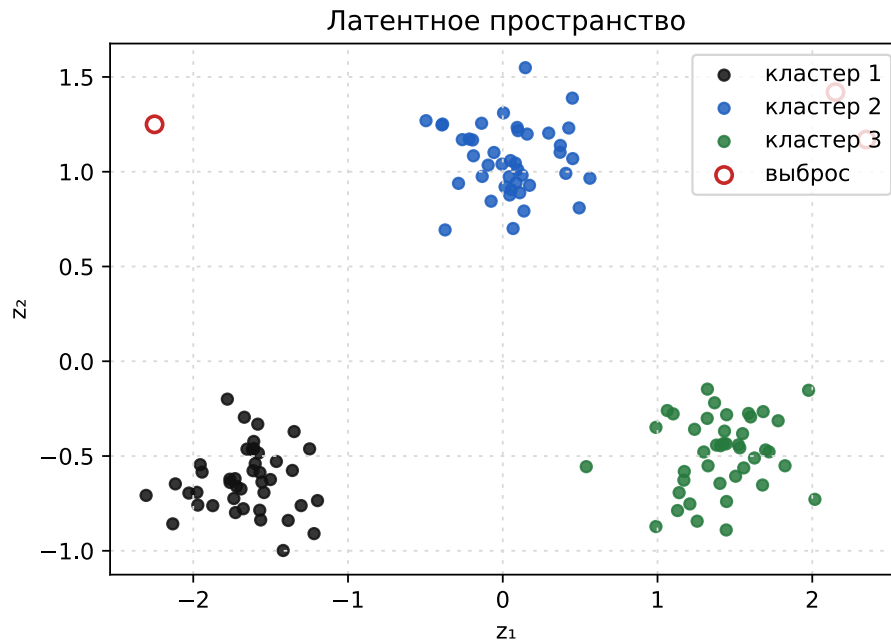


Рис. 11.2. Латентное пространство может выявлять кластеры и отделять нетипичные объекты.

11.9. Что важно вынести из этой лекции

Обучение без учителя изучает структуру данных без явных меток. Важную роль в нем играют латентные переменные, которые позволяют либо компактно описывать объекты, либо генерировать их из скрытого кода.

Генеративные модели составляют только часть обучения без учителя, но именно они особенно важны в современном глубоком обучении. Среди них выделяют вероятностные модели, умеющие не только порождать данные, но и оценивать их правдоподобие.

Оценка качества генеративной модели не сводится к одному числу. Нас одновременно интересуют:

- реалистичность примеров;
- полнота покрытия распределения;
- удобство латентного пространства;
- возможность вычислять правдоподобие;
- вычислительная эффективность генерации.

Именно из-за этих компромиссов разные архитектуры - GAN, VAE, нормализующие потоки и диффузионные модели (diffusion models) - занимают разные ниши.

11.10. Итог

Лекция 11 вводит обучение без учителя как следующий большой раздел после обучения с учителем и классического обучения нейросетей. Здесь ключевой вопрос уже не в том, как предсказать метку по входу, а в том, как раскрыть внутреннюю структуру данных, научиться компактно их представлять и синтезировать новые примеры.

Эта постановка особенно важна для современных генеративных моделей. В следующих лекциях естественным продолжением станут конкретные классы моделей, которые по-разному решают одну и ту же задачу: описать распределение сложных данных и научиться из него генерировать.

11.11. Проверочные вопросы

1. Чем обучение без учителя отличается от обучения с учителем на уровне постановки задачи?
2. Зачем в обучении без учителя вводят латентную переменную z ?
3. Чем генеративная модель отличается от вероятностной генеративной модели?
4. Почему функция потерь для вероятностной модели часто записывается как сумма отрицательных логарифмов правдоподобия?
5. Почему нельзя судить о качестве генеративной модели только по правдоподобию на обучающей выборке?
6. Что измеряют IS, FID, точность и полнота в контексте генеративных моделей?
7. Чем латентное пространство полезно для генерации новых объектов?
8. Как PCA и автоэнкодер строят компактное представление данных?
9. Почему ошибка восстановления может использоваться для поиска аномалий?
10. Почему разные генеративные архитектуры могут давать разные компромиссы между качеством, разнообразием и вычислительной стоимостью?

Источники

1. Simon J. D. Prince. *Understanding Deep Learning*. Chapter 14: Unsupervised learning. URL: <https://github.com/udlbook/udlbook>
2. <https://www.borealisai.com/en/blog/tutorial-5-variational-auto-encoders/>

Лекция 12. Сверточные сети для обработки изображений

Сверточные нейронные сети (Convolutional Neural Networks, CNN) строят представления изображений с помощью локальных фильтров, разделения параметров и иерархии признаков. В лекции рассматриваются математические свойства свертки, составные блоки CNN, классические архитектуры LeNet, AlexNet и VGG, а также связь сверточных представлений с задачами классификации, детекции и сегментации изображений.

12.1. Постановка проблемы

Изображение можно рассматривать как тензор

$$\mathbf{x} \in \mathbb{R}^{H \times W \times C}, \quad 12.1$$

где H и W задают пространственные размеры, а C — число каналов. Если такой тензор подать в полносвязный слой, то каждый выходной признак зависит от всех пикселей:

$$h_i = a \left[\beta_i + \sum_{j=1}^D \omega_{ij} x_j \right]. \quad 12.2$$

Для изображений это плохо по трем причинам.

1. Число параметров быстро растет вместе с разрешением изображения.
2. Полносвязный слой не использует локальность: соседние пиксели обычно сильнее связаны по смыслу, чем далекие.
3. Один и тот же объект может находиться в разных частях изображения, поэтому модель должна одинаково реагировать на локальный признак независимо от его положения.

Сверточные сети решают эти проблемы через локальные ядра и разделение параметров. Цель CNN — построить отображение

$$f_\phi : \mathbb{R}^{H \times W \times C} \rightarrow \mathcal{Y}, \quad 12.3$$

которое сохраняет полезную пространственную структуру и постепенно переводит пиксели в признаки высокого уровня: границы, текстуры, части объектов и целые объекты.

12.2. Инвариантность и эквивариантность

Для обработки изображений важны два свойства.

Инвариантность означает, что ответ модели не меняется при преобразовании входа:

$$f_{\phi} [\mathcal{T}[\mathbf{x}]] = f_{\phi}[\mathbf{x}]. \quad 12.4$$

Например, классификатор кошек должен распознавать кошку независимо от небольшого сдвига объекта в кадре.

Эквивариантность означает, что преобразование входа приводит к соответствующему преобразованию выхода:

$$f_{\phi} [\mathcal{T}[\mathbf{x}]] = \mathcal{T} [f_{\phi}[\mathbf{x}]]. \quad 12.5$$

Это свойство важно для плотных задач: если изображение сдвинулось, карта признаков или карта сегментации должна сдвинуться вместе с ним. Свертка является естественным механизмом для эквивариантной обработки.

12.3. Одномерная и двумерная свертка

В одномерном случае локальный фильтр длины 3 можно записать как

$$z_i = \omega_1 x_{i-1} + \omega_2 x_i + \omega_3 x_{i+1}. \quad 12.6$$

Если добавить смещение и нелинейность $a[\cdot]$, получаем сверточный слой:

$$h_i = a[\beta + \omega_1 x_{i-1} + \omega_2 x_i + \omega_3 x_{i+1}] = a \left[\beta + \sum_{j=1}^3 \omega_j x_{i+j-2} \right]. \quad 12.7$$

Ключевое отличие от полносвязного слоя состоит в том, что одни и те же параметры $\omega_1, \omega_2, \omega_3$ применяются ко всем позициям i . Поэтому фильтр учится находить один и тот же локальный шаблон в разных местах.

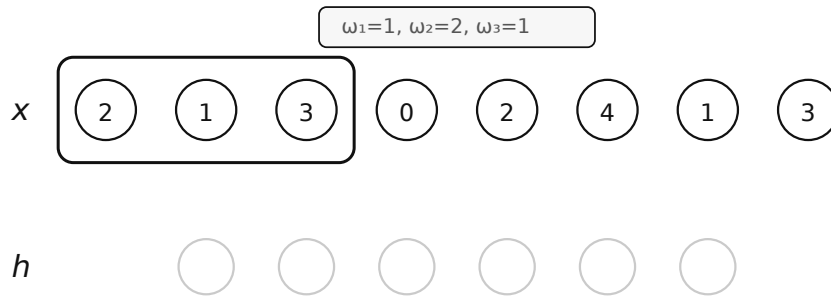
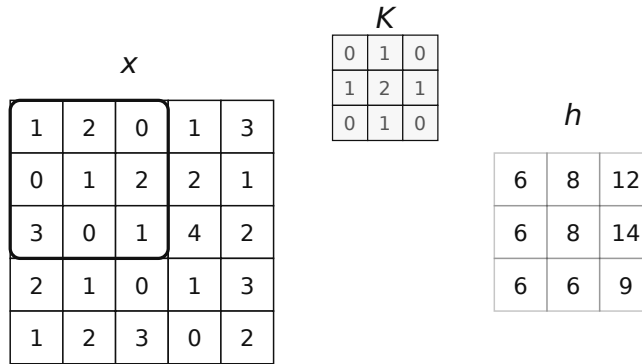


Рис. 12.1. Одномерная свертка как скользящее окно вдоль последовательности.

Для двумерного изображения с ядром 3×3 :

$$h_{ij} = a \left[\beta + \sum_{m=1}^3 \sum_{n=1}^3 \omega_{mn} x_{i+m-2, j+n-2} \right]. \quad 12.8$$

Рис. 12.2. Двумерная свертка: ядро 3×3 скользит по полю 5×5 .

В многоканальном случае фильтр имеет форму $K_h \times K_w \times C_{in}$, а слой содержит C_{out} таких фильтров. Для нечетных K_h и K_w обозначим радиусы ядра $r_h = (K_h - 1)/2$ и $r_w = (K_w - 1)/2$. Тогда для выходного канала c' :

$$h_{ijc'} = a \left[\beta_{c'} + \sum_{c=1}^{C_{in}} \sum_{u=-r_h}^{r_h} \sum_{v=-r_w}^{r_w} \omega_{uvcc'} x_{i+u, j+v, c} \right]. \quad 12.9$$

Эта формула показывает, как CNN одновременно использует локальность по пространству и смешивание информации между каналами.

12.4. Дополнение границ, шаг и рецептивное поле

Пусть вход имеет размер $H \times W$, ядро имеет размер $K_h \times K_w$, дополнение границ (padding) равно P_h, P_w , шаг окна (stride) равен S_h, S_w . Тогда пространственный размер выхода:

$$H_{\text{out}} = \left\lfloor \frac{H + 2P_h - K_h}{S_h} \right\rfloor + 1, \quad W_{\text{out}} = \left\lfloor \frac{W + 2P_w - K_w}{S_w} \right\rfloor + 1. \quad 12.10$$

Дополнение границ добавляет рамку вокруг изображения и позволяет сохранить размер карты признаков. **Шаг окна** задает перемещение окна и уменьшает разрешение выхода. Через несколько слоев каждый элемент глубокой карты признаков зависит от все большей области исходного изображения; эта область называется рецептивным полем (receptive field).

Если слой l имеет размер ядра k_l и эффективный шаг j_{l-1} относительно входа, то рецептивное поле можно рекуррентно оценить как

$$r_l = r_{l-1} + (k_l - 1)j_{l-1}, \quad j_l = j_{l-1}s_l. \quad 12.11$$

Эта связь объясняет, почему глубокая CNN может сочетать локальные фильтры с глобальным пониманием объекта.

12.5. Составные блоки сверточных сетей

Типичная CNN строится из повторяющихся блоков:

$$\text{свертка} \rightarrow \text{нормализация} \rightarrow \text{активация} \rightarrow \text{объединение}. \quad 12.12$$

Свертка извлекает локальные признаки. Нормализация стабилизирует распределение активаций. Нелинейность, например ReLU,

$$\text{ReLU}(z) = \max(0, z), \quad 12.13$$

позволяет строить сложные кусочно-линейные отображения. Объединение признаков (pooling) или свертка с шагом окна уменьшает пространственное разрешение и увеличивает устойчивость признаков к небольшим сдвигам.

Объединение признаков с максимизацией (max pooling) записывается как

$$h_{ijc} = \max_{(m,n) \in \mathcal{W}_{ij}} x_{mnc}, \quad 12.14$$

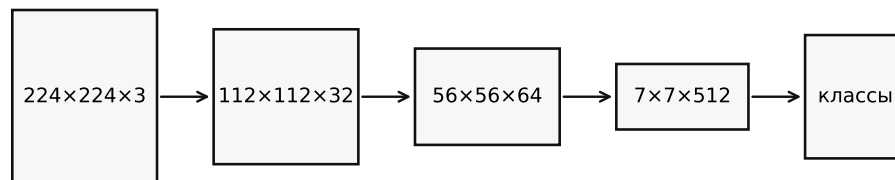
где $\mathcal{W}_{ij}$ — локальное окно. Объединение признаков с усреднением (average pooling) заменяет максимум средним значением:

$$h_{ijc} = \frac{1}{|\mathcal{W}_{ij}|} \sum_{(m,n) \in \mathcal{W}_{ij}} x_{mnc}. \quad 12.15$$

В конце классификационной сети пространственная карта признаков переводится в вектор классов. Для Q классов:

$$p(y = q \mid \mathbf{x}) = \frac{\exp(o_q)}{\sum_{r=1}^Q \exp(o_r)}, \quad L(\phi) = - \sum_{n=1}^N \log p(y_n \mid \mathbf{x}_n, \phi). \quad 12.16$$

Так сверточная архитектура связывается с общей задачей обучения с учителем (supervised learning).



пространственный размер уменьшается, число каналов обычно растёт

Рис. 12.3. В типичной CNN пространственный размер уменьшается, а число каналов признаков растёт.

12.6. Классические архитектуры: LeNet, AlexNet, VGG

LeNet показывает базовый принцип: несколько сверточных слоев и pooling постепенно уменьшают размер карт признаков, после чего полносвязная часть выполняет классификацию. Важная идея LeNet — не ручное выделение признаков, а совместное обучение фильтров и классификатора.

AlexNet масштабирует этот принцип на более сложные изображения: глубокая CNN, линейный выпрямитель (Rectified Linear Unit, ReLU), случайное выключение нейронов (dropout), обучение на графическом ускорителе (graphics processing unit, GPU) и большие данные. С математической точки зрения AlexNet сохраняет ту же логику:

$$\mathbf{x} \mapsto \mathbf{h}^{(1)} \mapsto \dots \mapsto \mathbf{h}^{(L)} \mapsto \mathbf{o} \mapsto p(y \mid \mathbf{x}). \quad 12.17$$

Отличие состоит в глубине, числе каналов и инженерной организации обучения.

VGG делает архитектуру более регулярной: она использует последовательности малых ядер 3×3 . Две свертки 3×3 имеют рецептивное поле 5×5 , а три — 7×7 , но между ними стоят нелинейности. Поэтому стек малых ядер часто выразительнее одного большого ядра.

Таблица 12.1. Сравнение классических сверточных архитектур.

Архитектура	Ключевая идея	Типичный блок	Исторический вклад
LeNet	Совместное обучение фильтров и классификатора	Conv $\rightarrow$ Pool $\rightarrow$ Conv $\rightarrow$ Pool $\rightarrow$ Linear	Показала практическую эффективность CNN для распознавания символов
AlexNet	Масштабирование CNN на большие изображения и данные	Conv/ReLU/Pool + Dropout + Linear	Закрепила глубокие CNN, ReLU и обучение на GPU
VGG	Регулярный стек малых ядер 3×3	Conv $3 \times 3 \rightarrow$ ReLU, повторяется много раз	Показала пользу глубины и простой повторяемой структуры

12.7. Распознавание образов

В задаче распознавания образов требуется определить класс изображения:

$$\hat{y} = \arg \max_q p(y = q \mid \mathbf{x}, \phi). \quad 12.18$$

Сверточная сеть здесь выполняет две функции:

1. строит признаки $\mathbf{h} = g_\phi(\mathbf{x})$;
2. решает задачу классификации по этим признакам.

На ранних слоях фильтры часто реагируют на локальные контрасты и границы, на средних — на текстуры и части объектов, на поздних — на высокоуровневые признаки. Это не жесткое правило, но полезная интерпретация иерархии признаков.

12.8. Семантическая сегментация

В семантической сегментации нужно назначить класс каждому пикселю:

$$\hat{y}_{ij} = \arg \max_q p(y_{ij} = q \mid \mathbf{x}, \phi). \quad 12.19$$

Поэтому выход модели — не один вектор классов, а карта

$$\mathbf{o} \in \mathbb{R}^{H \times W \times Q}. \quad 12.20$$

Для обучения используют пиксельную кросс-энтропию:

$$L(\phi) = - \sum_{i,j} \sum_{q=1}^Q \mathbb{1}[y_{ij} = q] \log p(y_{ij} = q \mid \mathbf{x}, \phi). \quad 12.21$$

Главная трудность — совместить глобальный контекст и точную локализацию границ. Поэтому сегментационные архитектуры часто имеют структуру кодировщик-декодировщик (encoder-decoder): кодировщик (encoder) уменьшает разрешение и извлекает признаки, декодировщик (decoder) восстанавливает пространственную карту.

12.9. Детекция объектов

В детекции требуется найти объекты и определить их классы. Выход содержит не только класс, но и ограничивающую рамку (bounding box):

$$b = (x_{\min}, y_{\min}, x_{\max}, y_{\max}). \quad 12.22$$

Типичная функция потерь объединяет классификационный и регрессионный члены:

$$L = L_{\text{cls}} + \lambda L_{\text{box}}. \quad 12.23$$

Для оценки качества локализации используют пересечение над объединением (Intersection over Union, IoU):

$$\text{IoU}(b, \hat{b}) = \frac{|b \cap \hat{b}|}{|b \cup \hat{b}|}. \quad 12.24$$

Сверточная часть модели строит карту признаков, а детекционная голова (detection head) предсказывает классы и координаты объектов для разных положений и масштабов.

12.10. Сегментация объектов

Сегментация объектов (instance segmentation) объединяет детекцию и пиксельную сегментацию. Требуется не только указать класс пикселя, но и разделить разные экземпляры одного класса:

$$x \mapsto \{(c_m, b_m, \mathbf{M}_m)\}_{m=1}^{N_{\text{obj}}}, \quad 12.25$$

где N_{obj} — число найденных объектов, c_m — класс, b_m — рамка, а $\mathbf{M}_m$ — маска объекта. Поэтому модель должна одновременно решать задачи классификации, локализации и построения маски.

Различие задач удобно держать в одной таблице.

Таблица 12.2. Форматы выходов в задачах компьютерного зрения.

Задача	Выход модели	Что нужно локализовать	Типичная функция потерь
Классификация	$p(y x)$	Только весь объект или сцена	Кросс-энтропия по классу
Семантическая сегментация	$p(y_{ij} x)$	Класс каждого пикселя	Пиксельная кросс-энтропия
Детекция	(c_m, b_m)	Рамки объектов	$L_{\text{cls}} + \lambda L_{\text{box}}$
Сегментация объектов	$(c_m, b_m, \mathbf{M}_m)$	Рамки и маски экземпляров	Классификация + рамки + маски

одна сверточная основа, разные выходные головы



Рис. 12.4. Разные задачи компьютерного зрения используют одну сверточную основу, но разные выходные головы.

12.11. Итог

1. Свертка использует локальность и разделение параметров, поэтому лучше подходит для изображений, чем полносвязная обработка пикселей.
2. CNN естественно поддерживает эквивариантность к сдвигам, а объединение признаков и иерархия признаков помогают получать частичную инвариантность.
3. Размеры карт признаков задаются ядром, дополнением границ и шагом окна.
4. Классические архитектуры LeNet, AlexNet и VGG показывают эволюцию от простых CNN к более глубоким и регулярным сетям.
5. Для классификации выходом является распределение по классам, для сегментации — распределение по классам в каждом пикселе, для детекции — классы и ограничивающие рамки.
6. Сегментация объектов требует отделять разные экземпляры одного класса и поэтому объединяет классификацию, локализацию и маски.

12.12. Проверочные вопросы

1. Почему полносвязный слой плохо масштабируется при работе с изображениями?
2. Чем инвариантность отличается от эквивариантности? Запишите соответствующие формулы.
3. Запишите формулу двумерной свертки с ядром 3×3 .

4. Как дополнение границ и шаг окна влияют на размер выходной карты признаков?
5. Что такое рецептивное поле и почему оно растет с глубиной сети?
6. Чем максимальное объединение признаков отличается от среднего?
7. Почему стек малых сверток 3×3 может быть предпочтительнее одного большого ядра?
8. Как записывается пиксельная кросс-энтропия для семантической сегментации?
9. Что измеряет IoU в задаче детекции объектов?
10. Чем семантическая сегментация отличается от сегментации объектов?

Источники

1. Prince S. J. D. *Understanding Deep Learning*. Chapter 10, Convolutional networks. <https://udlbook.github.io/udlbook/>; notebooks: <https://github.com/udlbook/udlbook>.
2. Zhang A., Lipton Z. C., Li M., Smola A. J. *Dive into Deep Learning*. Chapter 7, Convolutional Neural Networks; Chapter 8, Modern Convolutional Neural Networks; Chapter 14, Computer Vision. <https://github.com/d2l-ai/d2l-en>.
3. Dive into Deep Learning. Deep Convolutional Neural Networks (AlexNet). https://d2l.ai/chapter_convolutional-modern/alexnet.html.
4. Курс «Глубокое обучение». Архитектура LeNet. <https://deeppmachinelearning.ru/docs/Neural-networks/Convolutional-architectures/LeNet>.
5. Курс «Глубокое обучение». Модели AlexNet и ZFNet. <https://deeppmachinelearning.ru/docs/Neural-networks/Convolutional-architectures/AlexNet-ZFnet>.
6. Курс «Глубокое обучение». Модель VGG. <https://deeppmachinelearning.ru/docs/Neural-networks/Convolutional-architectures/VGG>.
7. Курс «Глубокое обучение». Семантическая сегментация. <https://deeppmachinelearning.ru/docs/Neural-networks/Semantic-segmentation>.

Лекция 13. Рекуррентные сети

Рекуррентные нейронные сети (Recurrent Neural Networks, RNN) предназначены для обработки последовательностей: текстов, временных рядов, аудиосигналов, траекторий и потоков событий. В лекции рассматриваются базовая RNN, обучение через развертку во времени, проблема затухающих градиентов, ячейки долгой краткосрочной памяти (Long Short-Term Memory, LSTM) и управляемого рекуррентного блока (Gated Recurrent Unit, GRU), а также RWKV как современная попытка совместить последовательную динамику RNN с масштабируемостью языковых моделей.

13.1. Постановка проблемы

Пусть наблюдается последовательность

$$\mathbf{x}_{1:T} = (\mathbf{x}_1, \mathbf{x}_2, \dots, \mathbf{x}_T). \quad 13.1$$

Задачи машинного обучения на последовательностях имеют разные формы:

1. классификация всей последовательности: $\mathbf{x}_{1:T} \mapsto y$;
2. прогноз следующего элемента: $\mathbf{x}_{1:t} \mapsto \mathbf{x}_{t+1}$;
3. разметка каждого элемента: $\mathbf{x}_{1:T} \mapsto y_{1:T}$;
4. преобразование одной последовательности в другую: $\mathbf{x}_{1:T} \mapsto y_{1:S}$.

Полносвязная сеть фиксированной размерности плохо подходит для таких задач: длина T может меняться, а порядок элементов важен. Рекуррентная сеть вводит скрытое состояние $\mathbf{h}_t$, которое переносит информацию из прошлого:

$$\mathbf{h}_t = f_\phi(\mathbf{x}_t, \mathbf{h}_{t-1}). \quad 13.2$$

Именно состояние делает модель последовательной: один и тот же блок применяется к каждому шагу времени, а параметры разделяются между шагами.

13.2. Базовая рекуррентная сеть

Классическая RNN задается формулами

$$\mathbf{h}_t = \tanh(\mathbf{W}_{xh}\mathbf{x}_t + \mathbf{W}_{hh}\mathbf{h}_{t-1} + \mathbf{b}_h), \quad 13.3$$

$$\mathbf{o}_t = \mathbf{W}_{hy} \mathbf{h}_t + \mathbf{b}_y, \quad p(y_t \mid \mathbf{x}_{1:t}) = \text{softmax}(\mathbf{o}_t). \quad 13.4$$

Если задача — языковое моделирование, то модель оценивает вероятность последовательности токенов:

$$P(w_{1:T}) = \prod_{t=1}^T P(w_t \mid w_{1:t-1}). \quad 13.5$$

Обучение сводится к минимизации отрицательного логарифма правдоподобия:

$$L(\phi) = - \sum_{t=1}^T \log P(w_t \mid w_{1:t-1}, \phi). \quad 13.6$$

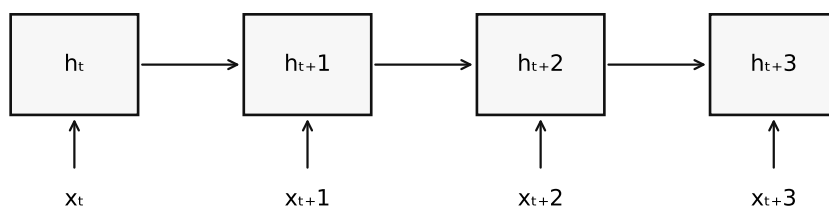
Разделение параметров означает, что одна и та же матрица $\mathbf{W}_{hh}$ применяется на каждом шаге. Это уменьшает число параметров и позволяет обрабатывать последовательности разной длины.

13.3. Развертка во времени и градиенты

Для обучения RNN используют **обратное распространение ошибки во времени** (backpropagation through time, BPTT). Это частный случай обратного режима автоматического дифференцирования, который был рассмотрен в лекции 6: рекуррентная сеть сначала разворачивается в обычный вычислительный граф длины T , а затем градиенты проходят по этому графу от поздних шагов к ранним.

$$\mathbf{h}_1 \rightarrow \mathbf{h}_2 \rightarrow \dots \rightarrow \mathbf{h}_T. \quad 13.7$$

один и тот же блок f_{θ} применяется на каждом шаге времени



BPTT распространяет градиент назад по развернутой цепочке

Рис. 13.1. Развертка RNN во времени превращает рекуррентное вычисление в обычный вычислительный граф.

Пусть суммарная функция потерь имеет вид

$$L(\phi) = \sum_{t=1}^T \ell_t(\mathbf{o}_t, y_t), \quad 13.8$$

где ℓ_t — ошибка на шаге t . Тогда БРТТ вычисляет градиент по параметрам как сумму вкладов всех временных шагов:

$$\frac{\partial L}{\partial \phi} = \sum_{t=1}^T \frac{\partial L}{\partial \mathbf{h}_t} \frac{\partial \mathbf{h}_t}{\partial \phi}. \quad 13.9$$

Градиент поздней ошибки по раннему состоянию содержит произведение якобианов:

$$\frac{\partial L}{\partial \mathbf{h}_t} = \frac{\partial L}{\partial \mathbf{h}_T} \prod_{\tau=t+1}^T \frac{\partial \mathbf{h}_\tau}{\partial \mathbf{h}_{\tau-1}}. \quad 13.10$$

Если нормы множителей меньше единицы, градиент затухает; если больше единицы, он взрывается. Поэтому простые RNN плохо удерживают долгосрочные зависимости.

На практике используют несколько приемов.

1. **Усеченное обратное распространение ошибки во времени** (truncated backpropagation through time, truncated БРТТ). Последовательность разбивают на окна длины $K < T$, и градиент не распространяют дальше начала текущего окна:

$$\frac{\partial L_{t-K+1:t}}{\partial \phi} = \sum_{\tau=t-K+1}^t \frac{\partial L_{t-K+1:t}}{\partial \mathbf{h}_\tau} \frac{\partial \mathbf{h}_\tau}{\partial \phi}. \quad 13.11$$

Так уменьшают стоимость обучения и ограничивают длину произведения якобианов. Цена приема — модель хуже получает обучающий сигнал от очень дальних зависимостей.

2. **Ограничение нормы градиента** (gradient clipping). Если полный градиент $\mathbf{g} = \nabla_{\phi} L$ слишком велик, его масштабируют:

$$\tilde{\mathbf{g}} = \mathbf{g} \cdot \min \left(1, \frac{c}{\|\mathbf{g}\|_2} \right), \quad 13.12$$

где c — заданный порог. Затем параметры обновляют уже по $\tilde{\mathbf{g}}$. Это не решает проблему затухающих градиентов, но защищает обучение от взрывов градиента.

3. **Управляемые рекуррентные ячейки** (gated recurrent units/cells). Такие ячейки вводят переменные-ворота (gates), которые регулируют запись, забывание и чтение состояния. В LSTM это видно по формуле памяти:

$$\mathbf{c}_t = \mathbf{f}_t \odot \mathbf{c}_{t-1} + \mathbf{i}_t \odot \tilde{\mathbf{c}}_t. \quad 13.13$$

Здесь $\mathbf{f}_t$ — ворота забывания (forget gate), $\mathbf{i}_t$ — входные ворота (input gate). Подробные формулы LSTM и GRU приведены в разделах 13.4 и 13.5; обзор этих блоков также есть в лекции 9.

4. **Остаточные связи** (residual connections) и **нормализация** (normalization). Остаточная связь добавляет тождественный путь для сигнала и градиента:

$$\mathbf{h}_t^{\text{res}} = \mathbf{h}_t + \mathbf{F}(\mathbf{h}_t, \phi). \quad 13.14$$

Производная такой конструкции содержит слагаемое от тождественного пути, поэтому градиенту проще проходить через глубокую композицию преобразований. Нормализация, например нормализация по слою (layer normalization, LayerNorm), стабилизирует масштаб скрытого состояния:

$$\text{LayerNorm}(\mathbf{h}) = \gamma \odot \frac{\mathbf{h} - \mu(\mathbf{h})}{\sqrt{\sigma^2(\mathbf{h}) + \varepsilon}} + \beta. \quad 13.15$$

Формулы нормализации и место LayerNorm в нейросетевых блоках были введены в лекции 9.

13.4. Долгая краткосрочная память

Долгая краткосрочная память (LSTM) вводит отдельное состояние памяти $\mathbf{c}_t$ и набор переменных-ворот. Типичная форма LSTM:

$$\mathbf{i}_t = \sigma(\mathbf{W}_{xi}\mathbf{x}_t + \mathbf{W}_{hi}\mathbf{h}_{t-1} + \mathbf{b}_i), \quad 13.16$$

$$\mathbf{f}_t = \sigma(\mathbf{W}_{xf}\mathbf{x}_t + \mathbf{W}_{hf}\mathbf{h}_{t-1} + \mathbf{b}_f), \quad 13.17$$

$$\mathbf{o}_t = \sigma(\mathbf{W}_{xo}\mathbf{x}_t + \mathbf{W}_{ho}\mathbf{h}_{t-1} + \mathbf{b}_o), \quad 13.18$$

$$\tilde{\mathbf{c}}_t = \tanh(\mathbf{W}_{xc}\mathbf{x}_t + \mathbf{W}_{hc}\mathbf{h}_{t-1} + \mathbf{b}_c), \quad 13.19$$

$$\mathbf{c}_t = \mathbf{f}_t \odot \mathbf{c}_{t-1} + \mathbf{i}_t \odot \tilde{\mathbf{c}}_t, \quad 13.20$$

$$\mathbf{h}_t = \mathbf{o}_t \odot \tanh(\mathbf{c}_t). \quad 13.21$$

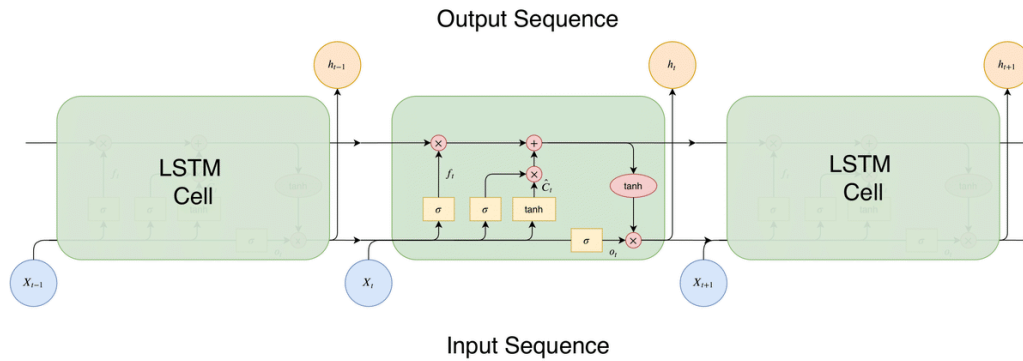


Рис. 13.2. Схема ячейки LSTM с воротами забывания, записи и выхода.

Здесь f_t управляет забыванием старой памяти, i_t — записью новой информации, o_t — тем, какая часть памяти попадет в скрытое состояние. Главная идея: градиент может проходить через аддитивное обновление c_t , а не только через многократное умножение на одну рекуррентную матрицу.

13.5. Управляемый рекуррентный блок

Управляемый рекуррентный блок (GRU) упрощает LSTM и объединяет память со скрытым состоянием. Основные уравнения:

$$z_t = \sigma(W_{xz}x_t + W_{hz}h_{t-1} + b_z), \quad 13.22$$

$$r_t = \sigma(W_{xr}x_t + W_{hr}h_{t-1} + b_r), \quad 13.23$$

$$\tilde{h}_t = \tanh(W_{xh}x_t + W_{hh}(r_t \odot h_{t-1}) + b_h), \quad 13.24$$

$$h_t = (1 - z_t) \odot h_{t-1} + z_t \odot \tilde{h}_t. \quad 13.25$$

Ворота обновления (update gate) z_t решают, насколько сильно заменить старое состояние новым, а ворота сброса (reset gate) r_t управляют тем, сколько прошлого использовать при построении кандидата $\tilde{h}_t$. GRU часто обучается быстрее LSTM и имеет меньше параметров, но выбор между ними зависит от задачи.

13.6. Рекуррентные сети в задачах преобразования последовательностей

В подходе кодировщик-декодировщик (encoder-decoder) кодировщик (encoder) переводит входную последовательность в состояние:

$$h_t^{\text{enc}} = f_\phi(x_t, h_{t-1}^{\text{enc}}), \quad 13.26$$

а декодировщик (decoder) порождает выход:

$$p(y_{1:S} \mid x_{1:T}) = \prod_{s=1}^S p(y_s \mid y_{1:s-1}, h_{1:T}^{\text{enc}}). \quad 13.27$$

Такой подход применялся в машинном переводе, распознавании речи и генерации текста. Ограничение классического кодировщика-декодировщика — необходимость сжать всю входную последовательность в фиксированное состояние. Механизм внимания, который затем стал основой трансформеров, снимает это ограничение: декодировщик может обращаться к разным состояниям кодировщика.

13.7. RWKV

RWKV — архитектура для языкового моделирования, которая стремится объединить сильные стороны RNN и трансформерного подхода (transformer approach). На уровне идеи она сохраняет рекуррентное состояние и поэтому может выполнять вывод (inference) за линейное время по длине последовательности, но использует обучаемые механизмы взвешивания прошлой информации, напоминающие внимание (attention).

Упрощенно рекуррентное обновление можно представить как

$$s_t = F_\phi(s_{t-1}, x_t), \quad o_t = G_\phi(s_t, x_t). \quad 13.28$$

В отличие от обычной RNN, состояние в RWKV проектируется так, чтобы эффективно агрегировать информацию о прошлом для больших языковых моделей. В учебном контексте RWKV важна как пример того, что рекуррентные идеи не исчезли после трансформеров: они продолжают развиваться в архитектурах, где важны длинные контексты и стоимость инференса.

13.8. Сравнение RNN, LSTM, GRU и RWKV

Базовая RNN проста, но плохо хранит долгосрочные зависимости. LSTM добавляет явную память и три механизма ворот (gate mechanisms). GRU уменьшает число ворот и часто дает близкое качество при меньшей сложности. RWKV переносит рекуррентную идею в масштаб языковых моделей и делает акцент на линейном по длине выводе.

С математической точки зрения все эти модели строят рекуррентное отображение

$$\mathbf{h}_t = f_\phi(\mathbf{x}_t, \mathbf{h}_{t-1}), \quad 13.29$$

но различаются тем, как устроена функция f_ϕ и как через нее проходит градиент.

13.9. Итог

1. Рекуррентные сети обрабатывают последовательности с помощью скрытого состояния и разделения параметров по времени.
2. Обучение RNN выполняется через развертку во времени и приводит к проблемам затухающих и взрывающихся градиентов.
3. LSTM вводит память c_t и переменные-ворота, чтобы лучше сохранять долгосрочную информацию.
4. GRU упрощает LSTM и объединяет память со скрытым состоянием.
5. Рекуррентные сети вида кодировщик-декодировщик применяются к преобразованию последовательностей, но ограничены сжатием контекста.
6. RWKV показывает современное направление: рекуррентная динамика может быть полезна даже в эпоху трансформеров, особенно для эффективного вывода.

13.10. Проверочные вопросы

1. Почему последовательности нельзя удобно обрабатывать обычной полносвязной сетью фиксированного размера?
2. Запишите формулу базового рекуррентного обновления $\mathbf{h}_t$.
3. Как языковая модель раскладывает вероятность $P(w_{1:T})$?
4. Почему при обратном распространении ошибки во времени (BPTT) возникает произведение якобианов?
5. В чем причина затухающих и взрывающихся градиентов в RNN?
6. Какую роль выполняют ворота забывания (forget gate), входные ворота (input gate) и выходные ворота (output gate) в LSTM?
7. Чем GRU отличается от LSTM?
8. Почему рекуррентная сеть вида кодировщик-декодировщик была важна для машинного перевода?
9. Какая идея связывает RWKV с рекуррентными сетями?
10. В каких задачах рекуррентные модели могут быть предпочтительны из-за стоимости вывода?

Источники

1. Zhang A., Lipton Z. C., Li M., Smola A. J. *Dive into Deep Learning*. Chapters 9-10, Recurrent Neural Networks and Modern Recurrent Neural Networks. <https://github.com/d2l-ai/d2l-en>.
2. Курс «Глубокое обучение». Рекуррентные сети. <https://deeplearning.ru/docs/Neural-networks/Recurrent-neural-nets>.
3. RWKV Language Model. <https://www.rwkv.com/>.
4. Prince S. J. D. *Understanding Deep Learning*. Chapter 11, Sequence modeling. <https://udlbook.github.io/udlbook/>; notebooks: <https://github.com/udlbook/udlbook>.

Лекция 14. Трансформеры

Трансформер (Transformer) — архитектура для обработки последовательностей и множеств, основанная на самовнимании (self-attention). В лекции рассматриваются представления запросов, ключей и значений (queries, keys, values), масштабированное скалярное произведение внимания (scaled dot-product attention), многоголовое внимание (multi-head attention), позиционное кодирование, блоки кодировщика и декодировщика (encoder/decoder blocks) и авторегрессионное моделирование. Главная линия изложения — как формулы внимания заменяют рекуррентную передачу состояния и позволяют эффективно строить контекстные представления токенов.

14.1. Постановка проблемы

В рекуррентной сети информация о прошлом передается через состояние

$$\mathbf{h}_t = f_\phi(\mathbf{x}_t, \mathbf{h}_{t-1}). \quad 14.1$$

Такой подход последователен по времени: чтобы получить $\mathbf{h}_t$, нужно вычислить предыдущие состояния. Кроме того, длинные зависимости проходят через цепочку преобразований. Трансформер решает другую задачу: для каждого элемента последовательности сразу построить представление, зависящее от всех релевантных элементов.

Пусть вход — матрица эмбедингов

$$\mathbf{X} = [\mathbf{x}_1, \dots, \mathbf{x}_N] \in \mathbb{R}^{D \times N}. \quad 14.2$$

Нужно построить

$$\mathbf{Y} = [\mathbf{y}_1, \dots, \mathbf{y}_N], \quad 14.3$$

где каждый $\mathbf{y}_n$ учитывает контекст всей последовательности. Самовнимание делает это через обучаемые веса связи между позициями.

14.2. Значения, запросы и ключи

Для каждого входного вектора строится значение (value):

$$\mathbf{v}_m = \beta_v + \Omega_v \mathbf{x}_m. \quad 14.4$$

Выход самовнимания для позиции n — взвешенная сумма значений:

$$\text{sa}_n[\mathbf{x}_1, \dots, \mathbf{x}_N] = \sum_{m=1}^N a[\mathbf{x}_m, \mathbf{x}_n] \mathbf{v}_m. \quad 14.5$$

Вес $a[\mathbf{x}_m, \mathbf{x}_n]$ показывает, насколько позиция n должна учитывать информацию из позиции m . Чтобы вычислить этот вес, вводятся запрос (query) и ключ (key):

$$\mathbf{q}_n = \beta_q + \Omega_q \mathbf{x}_n, \quad \mathbf{k}_m = \beta_k + \Omega_k \mathbf{x}_m. \quad 14.6$$

Тогда вес внимания:

$$a[\mathbf{x}_m, \mathbf{x}_n] = \text{softmax}_m[\mathbf{k}_m^T \mathbf{q}_n] = \frac{\exp(\mathbf{k}_m^T \mathbf{q}_n)}{\sum_{m'=1}^N \exp(\mathbf{k}_{m'}^T \mathbf{q}_n)}. \quad 14.7$$

Запрос задает вопрос текущей позиции, ключ описывает, на какие вопросы может отвечать позиция-источник, значение содержит передаваемую информацию.

14.3. Матричная форма самовнимания

Для всей последовательности:

$$\mathbf{V} = \beta_v \mathbf{1}^T + \Omega_v \mathbf{X}, \quad \mathbf{Q} = \beta_q \mathbf{1}^T + \Omega_q \mathbf{X}, \quad \mathbf{K} = \beta_k \mathbf{1}^T + \Omega_k \mathbf{X}. \quad 14.8$$

Самовнимание записывается как

$$\text{Sa}[\mathbf{X}] = \mathbf{V} \cdot \text{Softmax}(\mathbf{K}^T \mathbf{Q}). \quad 14.9$$

В современных трансформерах используют масштабирование:

$$\text{Sa}[\mathbf{X}] = \mathbf{V} \cdot \text{Softmax}\left(\frac{\mathbf{K}^T \mathbf{Q}}{\sqrt{D_q}}\right). \quad 14.10$$

Деление на $\sqrt{D_q}$ стабилизирует значения скалярных произведений: при большом D_q необработанные логиты (logits) могут стать слишком большими, softmax станет почти унитарным (one-hot), и градиенты ухудшатся.

14.4. Многоголовое внимание

Одна голова внимания строит один тип отношений между токенами. Многоголовое внимание использует H независимых голов:

$$\mathbf{V}_h = \beta_{vh} \mathbf{1}^T + \Omega_{vh} \mathbf{X}, \quad 14.11$$

$$\mathbf{Q}_h = \beta_{qh} \mathbf{1}^T + \Omega_{qh} \mathbf{X}, \quad 14.12$$

$$\mathbf{K}_h = \beta_{kh} \mathbf{1}^T + \Omega_{kh} \mathbf{X}. \quad 14.13$$

Для каждой головы:

$$\mathbf{Sa}_h[\mathbf{X}] = \mathbf{V}_h \cdot \text{Softmax} \left(\frac{\mathbf{K}_h^T \mathbf{Q}_h}{\sqrt{D_q}} \right). \quad 14.14$$

Затем результаты объединяются:

$$\mathbf{MhSa}[\mathbf{X}] = \Omega_c [\mathbf{Sa}_1[\mathbf{X}]^T, \dots, \mathbf{Sa}_H[\mathbf{X}]^T]^T. \quad 14.15$$

Разные головы могут специализироваться на разных зависимостях: локальных, синтаксических, дальних, позиционных или семантических.

14.5. Блок трансформера

Один блок трансформера обычно состоит из многоголового самовнимания, остаточных связей (residual connections), нормализации и позиционного многослойного перцептрона (Multilayer Perceptron, MLP):

$$\mathbf{U} = \mathbf{X} + \mathbf{MhSa}[\mathbf{X}], \quad 14.16$$

$$\mathbf{H} = \text{LayerNorm}[\mathbf{U}], \quad 14.17$$

$$\mathbf{m}_n = \mathbf{W}_2 \varphi(\mathbf{W}_1 \mathbf{h}_n + \mathbf{b}_1) + \mathbf{b}_2, \quad n = 1, \dots, N, \quad 14.18$$

$$\mathbf{Y} = \text{LayerNorm}[\mathbf{H} + \mathbf{M}]. \quad 14.19$$

Здесь $\mathbf{M}$ - матрица, составленная из столбцов $\mathbf{m}_n$. Внимание смешивает информацию между позициями, а MLP преобразует признаки внутри каждой позиции независимо от других позиций. Остаточные связи улучшают передачу градиента, нормализация стабилизирует обучение. LayerNorm применяется по координатам признаков внутри каждой позиции, а не по времени или по объектам мини-батча. В литературе также используют вариант pre-norm, где нормализация стоит перед вниманием и MLP; здесь записан post-norm-вариант для простоты.

14.6. Позиционное кодирование

Самовнимание без дополнительных признаков инвариантно к перестановке входных токенов: оно видит множество векторов, но не знает их порядок. Поэтому в трансформер добавляют позиционную информацию:

$$\tilde{\mathbf{x}}_n = \mathbf{x}_n + \boldsymbol{\pi}_n. \quad 14.20$$

В матричной форме:

$$\tilde{\mathbf{X}} = \mathbf{X} + \boldsymbol{\Pi}. \quad 14.21$$

Позиционные признаки могут быть фиксированными синусоидальными, обучаемыми абсолютными или относительными. Их задача — дать механизму внимания возможность различать порядок, расстояние и направление между токенами.

14.7. Кодировщик, декодировщик и причинная маска

Кодировщик (encoder) получает всю последовательность и строит контекстные представления. Он подходит для классификации текста, извлечения признаков и задач понимания.

Декодировщик (decoder) генерирует последовательность слева направо. Для языковой модели используется авторегрессионное разложение:

$$P(t_1, t_2, \dots, t_N) = P(t_1) \prod_{n=2}^N P(t_n \mid t_1, \dots, t_{n-1}). \quad 14.22$$

Чтобы токен t_n не видел будущие токены, в внимание добавляют причинную маску (causal mask):

$$a_{mn} = 0 \quad \text{при } m > n. \quad 14.23$$

Это превращает самовнимание в допустимый механизм авторегрессионной генерации.

Архитектура кодировщик-декодировщик (encoder-decoder) использует кодировщик для входной последовательности и декодировщик для выходной. В перекрестном внимании (cross-attention) запросы поступают из декодировщика, а ключи и значения — из кодировщика.

14.8. Сложность и ограничения

Главное вычислительное ограничение самовнимания — матрица $\mathbf{K}^T \mathbf{Q} \in \mathbb{R}^{N \times N}$. Поэтому сложность по длине последовательности:

$$O(N^2 D). \quad 14.24$$

Это делает стандартный трансформер мощным, но дорогим на длинных контекстах. Расширения трансформеров пытаются уменьшить эту стоимость через разреженное внимание (sparse attention), локальное внимание (local attention), механизмы памяти, линейные аппроксимации или специальные позиционные схемы.

14.9. Трансформеры в обработке текста

В задачах обработки естественного языка трансформер не получает исходную строку напрямую. Типичный конвейер состоит из трех шагов:

1. текст разбивается на токены;
2. каждый токен заменяется обучаемым вектором-эмбеддингом;
3. матрица эмбеддингов проходит через модель кодировщика, декодировщика или кодировщика-декодировщика.

Если текстовая последовательность после токенизации имеет длину N , то вход трансформера снова записывается как матрица

$$\mathbf{X} = [\mathbf{x}_1, \dots, \mathbf{x}_N] \in \mathbb{R}^{D \times N}. \quad 14.25$$

Отличие от абстрактной схемы самовнимания состоит в том, что теперь столбец $\mathbf{x}_n$ соответствует не произвольному элементу последовательности, а слову, знаку пунктуации, фрагменту слова или специальному служебному токenu.

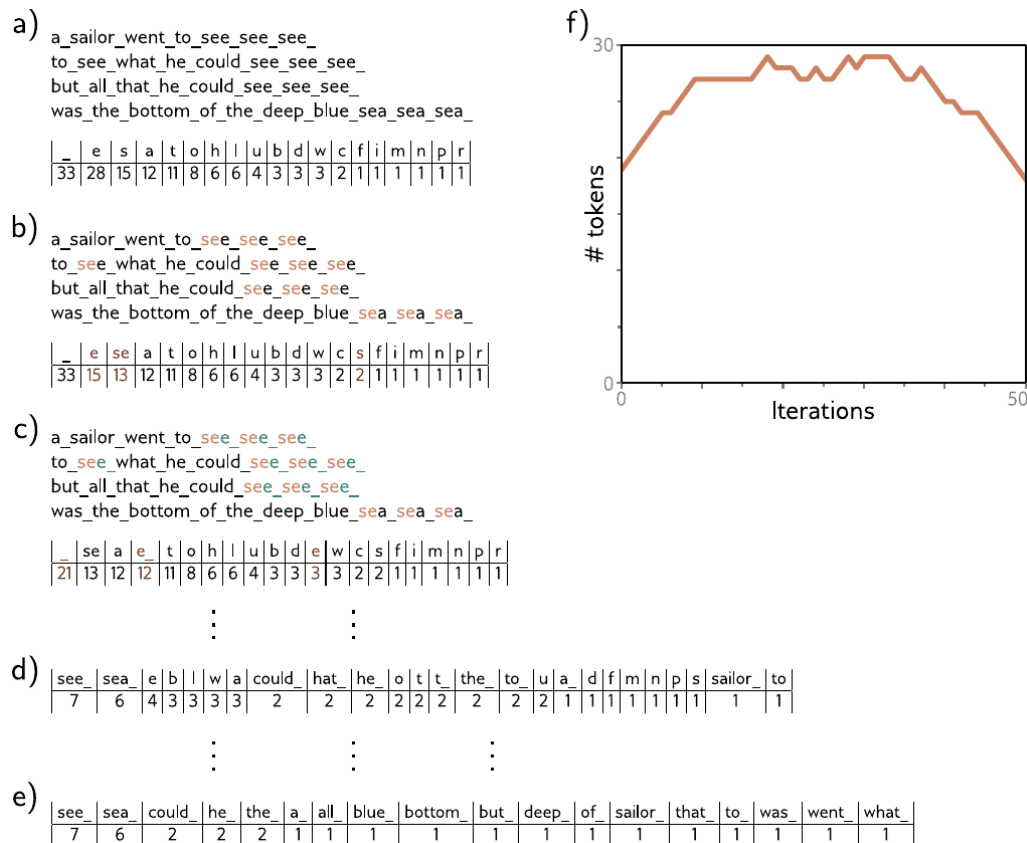


Рис. 14.1. Подсловная токенизация в UDL, Figure 12.8.

Иллюстрации главы 12 книги *Understanding Deep Learning* доступны на сайте книги: <https://udlbook.github.io/udlbook/>.

14.9.1. Токенизация

Наивный словарь из полных слов неудобен: в нем неизбежно появляются неизвестные имена, нужно отдельно кодировать пунктуацию, а формы вроде *walk, walks, walked, walking* оказываются независимыми токенами. Словарь из отдельных символов решает проблему неизвестных слов, но делает последовательность слишком длинной и заставляет сеть заново учить устойчивые фрагменты слов.

На практике используют компромисс: подсловную токенизацию. Словарь содержит символы, частые слова и часто встречающиеся фрагменты слов. В кодировании парами байтов (Byte Pair Encoding, BPE) алгоритм начинает с мелких токенов и жадно объединяет наиболее частые соседние пары. Процесс останавливается, когда размер словаря достигает заданного значения [12].

Такой подход важен для обучения трансформеров:

- редкие слова можно собрать из фрагментов;
- пунктуация и регистр могут быть отдельными признаками;
- длина последовательности остается умеренной;

- один и тот же словарь можно использовать на больших корпусах без ручной разметки.

14.9.2. Эмбединги токенов

Пусть $\mathcal{V}$ — словарь токенов, а $|\mathcal{V}|$ — его размер. Каждому токenu ставится в соответствие унитарный вектор (one-hot vector), а вся последовательность кодируется матрицей

$$\mathbf{T} \in \mathbb{R}^{|\mathcal{V}| \times N}. \quad 14.26$$

n -й столбец $\mathbf{T}$ содержит единицу в строке, соответствующей n -му токenu. Обучаемая матрица эмбедингов

$$\Omega_e \in \mathbb{R}^{D \times |\mathcal{V}|} \quad 14.27$$

преобразует унитарное представление в плотные векторы:

$$\mathbf{X} = \Omega_e \mathbf{T}. \quad 14.28$$

Матрица Ω_e является параметром модели и обучается вместе с остальными весами. Если $D = 1024$, а $|\mathcal{V}| = 30000$, то только таблица эмбедингов содержит десятки миллионов параметров. Это не побочная деталь, а часть модели: именно здесь дискретные токены получают непрерывное представление, с которым дальше работают attention-слои.

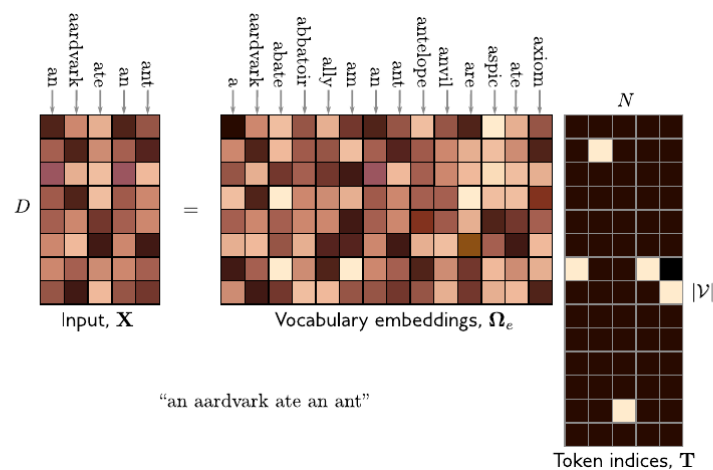


Рис. 14.2. Построение матрицы входных эмбедингов в UDL, Figure 12.9.

14.10. Модели кодировщика для понимания текста

Модель кодировщика получает всю токенизированную последовательность сразу. В каждом слое самовнимания любой токен может учитывать левый и правый контекст. Поэтому кодировщик хорошо

подходит для задач, где нужно построить контекстное представление уже данного текста:

- классификация текста;
- классификация каждого слова или токена;
- извлечение именованных сущностей;
- поиск ответа как фрагмента текста;
- построение универсального признакового представления для последующей донастройки.

BERT — типичный пример модели кодировщика. В книге Prince он описан как модель с большим словарем токенов, высокоразмерными эмбедингами и последовательностью слоев трансформера. Главная идея не в конкретных размерах, а в двухэтапном обучении: сначала самостоятельное предобучение (self-supervised pre-training) на большом корпусе, затем донастройка (fine-tuning) на целевой размеченной задаче.

14.10.1. Предобучение через маскированное языковое моделирование

При предобучении часть входных токенов заменяется специальным токеном `<mask>`. Кодировщик видит весь контекст вокруг пропуска и должен восстановить исходный токен. Если выходной вектор для замаскированной позиции равен $\mathbf{z}_n$, то линейный классификатор по словарю задает распределение

$$\mathbf{p}_n = \text{softmax}(\mathbf{\Omega}_o \mathbf{z}_n + \beta_o). \quad 14.29$$

Для истинного токена t_n вклад в функцию потерь имеет вид

$$L_n = -\log p_n(t_n). \quad 14.30$$

Такой сигнал обучения не требует ручных меток: метка берется из самого текста. При этом модель вынуждена использовать двусторонний контекст, потому что правильный токен часто определяется словами и слева, и справа.

14.10.2. Донастройка для прикладных задач

После предобучения к кодировщику добавляют небольшой выходной слой под конкретный формат ответа. Для классификации текста используют специальный токен `<cls>`: его выходной вектор считается представлением всей последовательности. Например, для бинарной классификации

$$\hat{y} = \sigma(\mathbf{w}^T \mathbf{z}_{\text{cls}} + b). \quad 14.31$$

Для классификации слов выходной слой применяется к каждому токenu отдельно. Для поиска текстового фрагмента, как в задаче question answering, модель может предсказывать две вероятности для каждой позиции: вероятность начала и вероятность конца ответа.

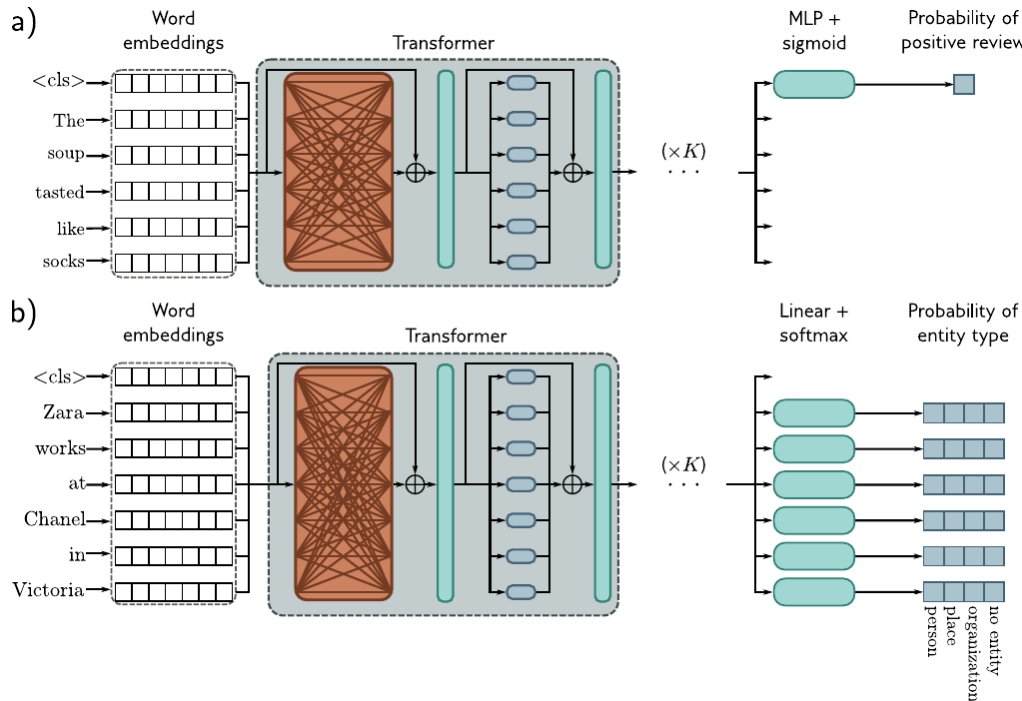


Рис. 14.3. Предобучение и донастройка encoder-модели в UDL, Figures 12.10-12.11.

14.11. Модели декодировщика для генерации текста

Модель декодировщика решает другую задачу: она строит распределение следующего токена по уже имеющемуся префиксу. Для последовательности токенов $t_1, \dots, t_N$ используется авторегрессионное разложение

$$P(t_1, \dots, t_N) = P(t_1) \prod_{n=2}^N P(t_n | t_1, \dots, t_{n-1}). \quad 14.32$$

Во время обучения модель получает последовательность токенов, но причинная маска запрещает позиции n смотреть на будущие токены $t_{n+1}, \dots, t_N$. Поэтому для каждой позиции можно вычислить вклад в функцию потерь одновременно, не нарушая авторегрессионную постановку:

$$L = - \sum_{n=1}^N \log P(t_n | t_1, \dots, t_{n-1}). \quad 14.33$$

В реализации с teacher forcing вход обычно сдвигают: модель получает префикс до позиции $n - 1$ и предсказывает токен t_n , либо эквивалентно выход позиции n используется для предсказания следующего токена t_{n+1} . После слоев трансформера выходной вектор позиции переводится линейным слоем в логиты размера $|\mathcal{V}|$, а softmax превращает их в распределение предсказываемого токена.

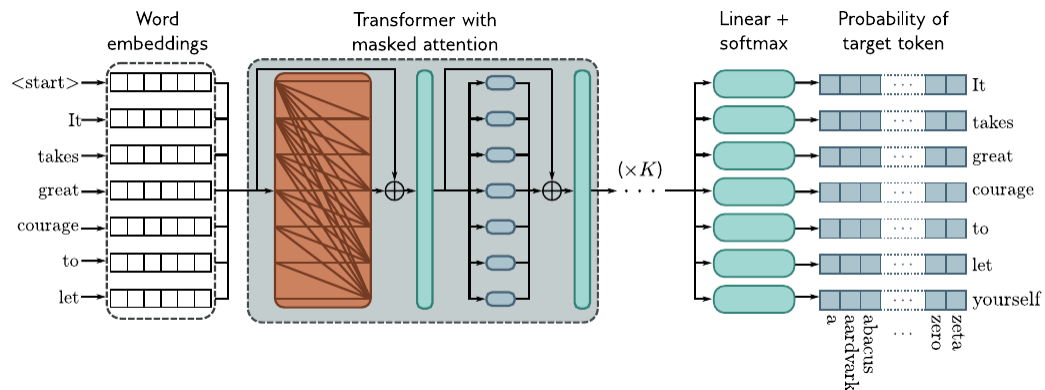


Рис. 14.4. Обучение модели декодировщика с маскированным самовниманием в UDL, Figure 12.12.

14.11.1. Генерация и стратегии декодирования

При генерации декодировщик запускается итеративно. Сначала задается начальный контекст, затем модель выдает распределение следующего токена. Выбранный токен добавляется к контексту, и процесс повторяется.

Самый простой вариант — выбирать токен с максимальной вероятностью. Но жадный выбор может рано завести модель в плохое продолжение. Поэтому используют стратегии декодирования:

- лучевой поиск (beam search) хранит несколько наиболее вероятных продолжений;
- выборка top- k (top- k sampling) выбирает следующий токен случайно, но только среди k наиболее вероятных;
- вероятностное сэмплирование сохраняет разнообразие текста и делает декодировщик генеративной моделью.

Большие языковые модели (Large Language Models, LLM) масштабируют эту схему: увеличивают число слоев, размер эмбеддингов, число голов, длину контекста и объем корпуса. В результате декодировщик может выполнять часть задач без донастройки, если задача задана прямо в контексте несколькими примерами. Это называют обучением по нескольким примерам (few-shot learning) или обучением в контексте (in-context learning), но качество такого поведения зависит от данных, формулировки запроса и масштаба модели.

14.12. Модели кодировщика-декодировщика для перевода и условной генерации

В задачах преобразования последовательности в последовательность (sequence-to-sequence) входная и выходная строки различны. Машинный перевод — базовый пример: кодировщик читает предложение на исходном языке, декодировщик генерирует предложение на целевом языке.

Пусть кодировщик построил матрицу выходных представлений

$$\mathbf{X}_{\text{enc}} \in \mathbb{R}^{D \times N_{\text{enc}}}, \quad 14.34$$

а декодировщик на текущем шаге имеет представления

$$\mathbf{X}_{\text{dec}} \in \mathbb{R}^{D \times N_{\text{dec}}}. \quad 14.35$$

В перекрестном внимании запросы вычисляются из представлений декодировщика, а ключи и значения — из представлений кодировщика:

$$\mathbf{Q} = \Omega_q \mathbf{X}_{\text{dec}}, \quad \mathbf{K} = \Omega_k \mathbf{X}_{\text{enc}}, \quad \mathbf{V} = \Omega_v \mathbf{X}_{\text{enc}}. \quad 14.36$$

Тогда декодировщик одновременно учитывает два источника информации: уже сгенерированный префикс на целевом языке через маскированное самовнимание (masked self-attention) и исходное предложение через перекрестное внимание:

$$\mathbf{Y}_{\text{cross}} = \mathbf{V} \cdot \text{Softmax} \left(\frac{\mathbf{K}^T \mathbf{Q}}{\sqrt{D_q}} \right). \quad 14.37$$

Именно эта схема позволяет каждому генерируемому токеноу зависеть от всех токенов исходной фразы, а не только от предыдущих токенов перевода.

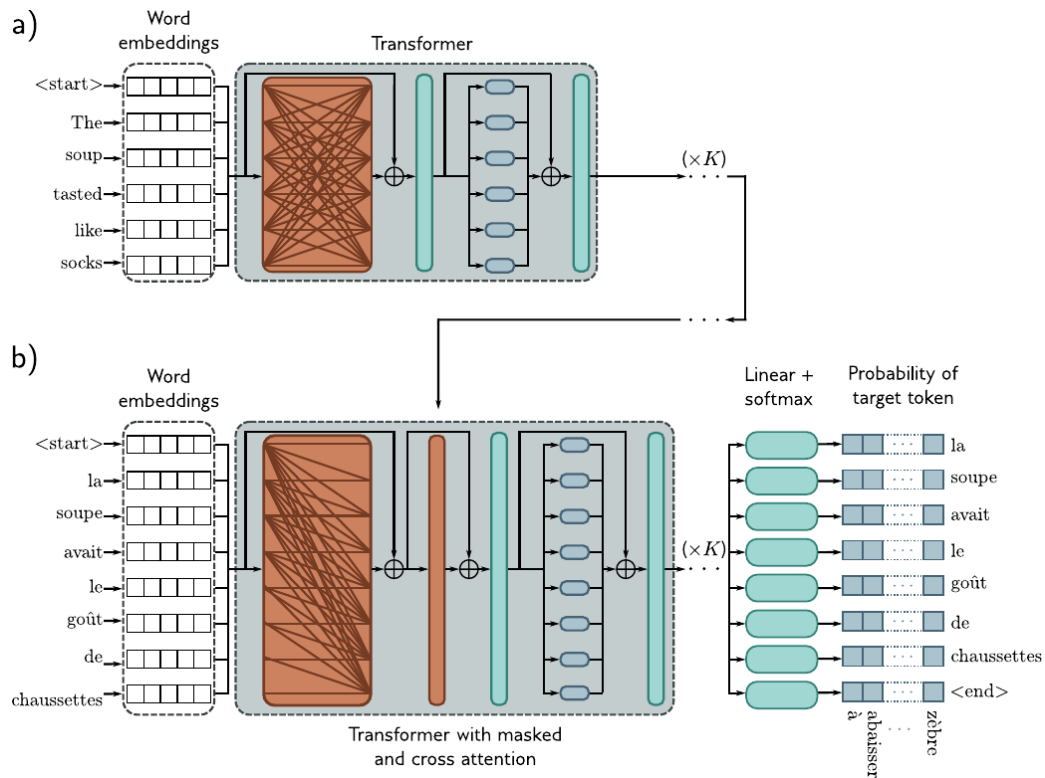


Рис. 14.5. Архитектура кодировщика-декодировщика и перекрестное внимание в UDL, Figures 12.13-12.14.

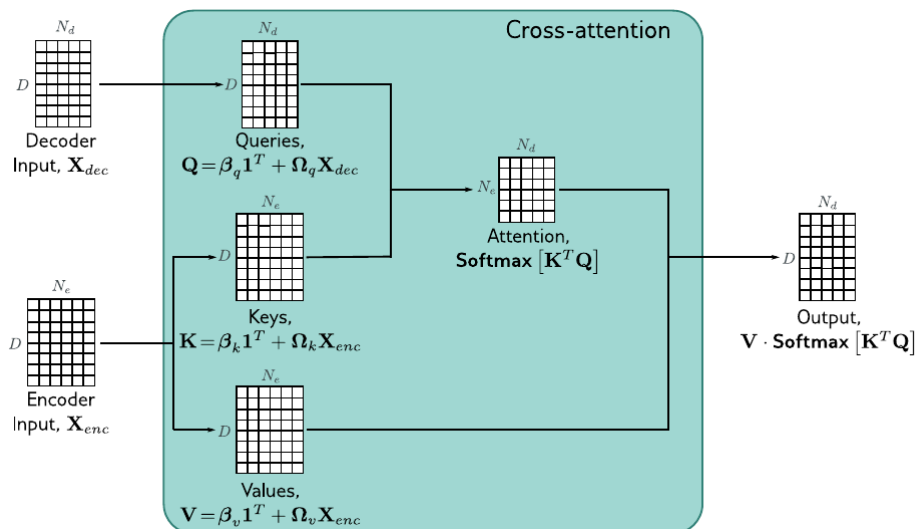


Рис. 14.6. Перекрестное внимание как отдельный блок слоя декодировщика в UDL, Figure 12.14.

14.13. Длинные текстовые последовательности

В кодировщике каждый токен взаимодействует с каждым другим токеном, поэтому матрица внимания имеет размер $N \times N$. В декодировщике причинная маска оставляет примерно половину взаимодействий, но порядок роста остается квадратичным:

$$O(N^2D).$$

14.38

Для длинных документов, диалогов и больших контекстов это становится главным ограничением. Один класс решений — разреживать матрицу взаимодействий:

- локальное окно внимания, где токен видит только соседние позиции;
- разреженное внимание с пропусками (dilated attention), где связи идут с пропусками и быстрее расширяют рецептивное поле (receptive field);
- специальные глобальные токены, которые соединяются со всеми позициями;
- выбранные позиции, которым разрешено видеть всю последовательность.

Компромисс здесь неизбежен: полное самовнимание дает прямые связи между всеми токенами, но дорого; разреженное внимание дешевле, но требует архитектурного решения о том, какие дальние зависимости можно сохранить.

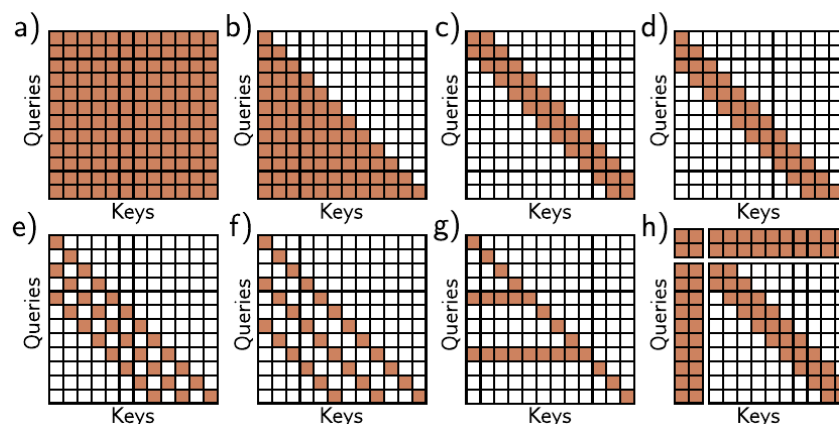


Рис. 14.7. Матрицы взаимодействий для длинных последовательностей в UDL, Figure 12.15.

14.14. Применения трансформеров за пределами текста

Та же математическая схема переносится на изображения, аудио, графы и мультимодальные данные. Для изображения его можно разбить на фрагменты (patches) и рассматривать как последовательность визуальных токенов. Тогда формулы внимания остаются теми же, а меняется только способ получения входных токенов и позиционных признаков.

14.15. Итог

1. Самовнимание строит контекстное представление позиции как взвешенную сумму значений других позиций.
2. Веса внимания вычисляются через совместимость запроса и ключа.
3. Масштабирование на $\sqrt{D_q}$ стабилизирует softmax.
4. Многоголовое внимание позволяет параллельно учитывать разные типы зависимостей.
5. Блок трансформера объединяет внимание, MLP, остаточные связи и нормализацию.
6. Позиционное кодирование необходимо, потому что самовнимание само по себе не знает порядок токенов.
7. Трансформер-декодировщик использует причинную маску и авторегрессионное разложение вероятности.
8. Для обработки естественного языка (Natural Language Processing, NLP) текст сначала переводится в токены и обучаемые эмбединги.
9. Модели кодировщика подходят для понимания текста и донастройки, модели декодировщика — для авторегрессионной генерации.
10. Модели кодировщика-декодировщика решают задачи преобразования последовательности в последовательность через перекрестное внимание.
11. Основное ограничение стандартного внимания — квадратичная сложность по длине последовательности.

14.16. Проверочные вопросы

1. Почему трансформеры не нуждаются в рекуррентном состоянии для передачи контекста?
2. Что означают запрос, ключ и значение?
3. Запишите формулу веса внимания через softmax от $\mathbf{k}_m^T \mathbf{q}_n$.
4. Почему в масштабированном скалярном произведении внимания делят на $\sqrt{D_q}$?
5. Как записывается многоголовое внимание?
6. Какую роль выполняет MLP внутри блока трансформера?
7. Почему самовнимание требует позиционного кодирования?

8. Что такое причинная маска и зачем она нужна модели декодировщика?
9. Запишите авторегрессионное разложение $P(t_1, \dots, t_N)$.
10. Чем модели кодировщика, декодировщика и кодировщика-декодировщика отличаются в задачах обработки текста?

Источники

1. Prince S. J. D. *Understanding Deep Learning*. Chapter 12, Transformers, sections 12.5-12.9. <https://udlbook.github.io/udlbook/>; notebooks and figures: <https://github.com/udlbook/udlbook>.
2. Zhang A., Lipton Z. C., Li M., Smola A. J. *Dive into Deep Learning*. Chapter 11, Attention Mechanisms and Transformers. <https://github.com/d2l-ai/d2l-en>.
3. Dive into Deep Learning. Queries, Keys, and Values. https://d2l.ai/chapter_attention-mechanisms-and-transformers/queries-keys-values.html.
4. Dive into Deep Learning. Attention Scoring Functions. https://d2l.ai/chapter_attention-mechanisms-and-transformers/attention-scoring-functions.html.
5. Prince S. Tutorial #14: Transformers I: Introduction. RBC Borealis. <https://rbcborealis.com/research-blogs/tutorial-14-transformers-i-introduction/>.
6. Prince S. Tutorial #16: Transformers II: Extensions. RBC Borealis. <https://rbcborealis.com/research-blogs/tutorial-16-transformers-ii-extensions/>.
7. Курс «Глубокое обучение». Трансформер. <https://deeppmachinelearning.ru/docs/Neural-networks/Transformer>.

Лекция 15. Обучение с подкреплением

Обучение с подкреплением (Reinforcement Learning, RL) изучает задачи, где агент выбирает действия, получает вознаграждения и должен максимизировать суммарную полезность во времени. В лекции рассматриваются марковские процессы принятия решений, функции ценности, уравнения Беллмана, динамическое программирование, методы Монте-Карло (Monte Carlo), методы временных разностей (temporal-difference methods), Q-обучение (Q-learning), глубокое Q-обучение (Deep Q-learning) и градиент политики (policy gradient).

15.1. Постановка проблемы

В обучении с учителем (supervised learning) обучающая пара имеет вид (x, y) , а функция потерь напрямую сравнивает предсказание с правильным ответом. В обучении с подкреплением правильное действие обычно неизвестно. Агент находится в состоянии s_t , выбирает действие a_t , получает награду r_t и переходит в новое состояние s_{t+1} .

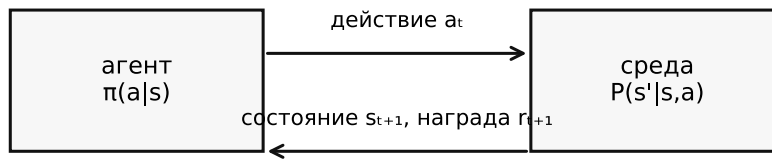
Цель — найти политику

$$\pi(a \mid s) = P(a_t = a \mid s_t = s), \quad 15.1$$

которая максимизирует ожидаемую дисконтированную сумму наград:

$$G_t = \sum_{k=0}^{\infty} \gamma^k r_{t+k+1}, \quad 0 \leq \gamma < 1. \quad 15.2$$

Сложность состоит в том, что действие влияет не только на немедленную награду, но и на будущие состояния.



цель: максимизировать $E[G_t]$

Рис. 15.1. Цикл взаимодействия агента и среды в обучении с подкреплением.

15.2. Марковский процесс принятия решений

Марковский процесс принятия решений задается пятеркой

$$(\mathcal{S}, \mathcal{A}, P, r, \gamma), \quad 15.3$$

где $\mathcal{S}$ — множество состояний, $\mathcal{A}$ — множество действий, $P(s' | s, a)$ — вероятность перехода, $r(s, a)$ — ожидаемая награда, γ — коэффициент дисконтирования.

Марковское свойство означает:

$$P(s_{t+1} | s_t, a_t, s_{t-1}, a_{t-1}, \dots) = P(s_{t+1} | s_t, a_t). \quad 15.4$$

Политика превращает марковский процесс принятия решений (Markov Decision Process, MDP) в случайный процесс: действия выбираются по $\pi(a | s)$, а среда переходит по $P(s' | s, a)$.

15.3. Функции ценности

Функция ценности состояния:

$$v_\pi(s) = \mathbb{E}_\pi[G_t | s_t = s]. \quad 15.5$$

Функция ценности действия:

$$q_\pi(s, a) = \mathbb{E}_\pi[G_t | s_t = s, a_t = a]. \quad 15.6$$

Оптимальные функции:

$$v^*(s) = \max_\pi \mathbb{E}[G_t | s_t = s, \pi], \quad 15.7$$

$$q^*(s, a) = \max_{\pi} \mathbb{E}[G_t \mid s_t = s, a_t = a, \pi]. \quad 15.8$$

Если известна q^* , оптимальную политику можно выбрать жадно:

$$\pi^*(s_t) \in \arg \max_a q^*(s_t, a). \quad 15.9$$

Если нужна вероятностная запись детерминированной политики, то можно записать

$$\pi^*(a \mid s_t) = \begin{cases} 1 & a = \pi^*(s_t) \\ 0 & a \neq \pi^*(s_t). \end{cases} \quad 15.10$$

Именно поэтому многие алгоритмы RL сводятся к оцениванию функций ценности.

15.4. Уравнения Беллмана

Связь между v_{π} и q_{π} :

$$v_{\pi}(s_t) = \sum_{a_t} \pi(a_t \mid s_t) q_{\pi}(s_t, a_t). \quad 15.11$$

Функция действия раскладывается на немедленную награду и ценность следующего состояния:

$$q_{\pi}(s_t, a_t) = r(s_t, a_t) + \gamma \sum_{s_{t+1}} P(s_{t+1} \mid s_t, a_t) v_{\pi}(s_{t+1}). \quad 15.12$$

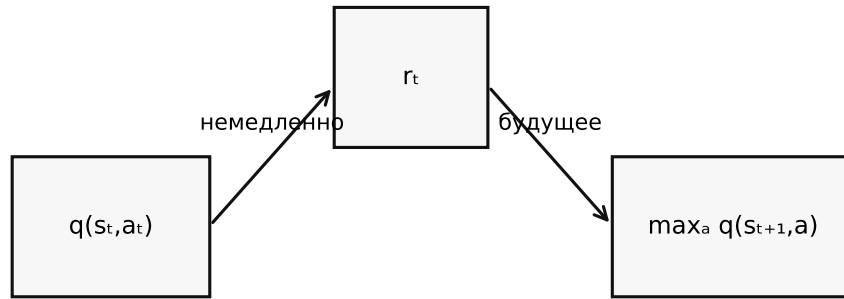
Отсюда уравнение Беллмана для v_{π} :

$$v_{\pi}(s_t) = \sum_{a_t} \pi(a_t \mid s_t) \left(r(s_t, a_t) + \gamma \sum_{s_{t+1}} P(s_{t+1} \mid s_t, a_t) v_{\pi}(s_{t+1}) \right). \quad 15.13$$

Для оптимальной функции действия:

$$q^*(s_t, a_t) = r(s_t, a_t) + \gamma \sum_{s_{t+1}} P(s_{t+1} \mid s_t, a_t) \max_a q^*(s_{t+1}, a). \quad 15.14$$

Эта рекуррентная структура — центральная математика обучения с подкреплением.



цель TD: $y_t = r_t + \gamma \max_a q(s_{t+1}, a)$

Рис. 15.2. Обновление Беллмана связывает текущую ценность с немедленной наградой и будущей ценностью.

15.5. Динамическое программирование

Если модель среды $P(s' | s, a)$ и награда известны, можно использовать динамическое программирование.

Оценивание политики:

$$v(s_t) \leftarrow \sum_{a_t} \pi(a_t | s_t) \left(r(s_t, a_t) + \gamma \sum_{s_{t+1}} P(s_{t+1} | s_t, a_t) v(s_{t+1}) \right). \quad 15.15$$

Улучшение политики:

$$\pi(s_t) \leftarrow \arg \max_{a_t} \left[r(s_t, a_t) + \gamma \sum_{s_{t+1}} P(s_{t+1} | s_t, a_t) v(s_{t+1}) \right]. \quad 15.16$$

Недостаток: в реальных задачах модель среды часто неизвестна, а пространство состояний слишком велико.

15.6. Методы Монте-Карло и временных разностей

Методы Монте-Карло оценивают ценность по завершённым траекториям:

$$q(s, a) \approx \frac{1}{N(s, a)} \sum_{i: s_i = s, a_i = a} G_i. \quad 15.17$$

Они не требуют модели среды, но ждут окончания эпизода и могут иметь высокую дисперсию.

Методы временных разностей обновляют оценку по одному шагу. Для алгоритма SARSA (State-Action-Reward-State-Action):

$$q(s_t, a_t) \leftarrow q(s_t, a_t) + \alpha(r_t + \gamma q(s_{t+1}, a_{t+1}) - q(s_t, a_t)). \quad 15.18$$

Величина

$$\delta_t = r_t + \gamma q(s_{t+1}, a_{t+1}) - q(s_t, a_t) \quad 15.19$$

называется ошибкой временной разности (temporal-difference error, TD-ошибка). Она показывает, насколько новая оценка отличается от текущей.

Таблица 15.1. Сравнение базовых методов оценки ценности.

Метод	Целевое значение	Когда обновляется	Свойство
Монте-Карло	G_t	После завершения эпизода	Без модели среды, но высокая дисперсия
SARSA	$r_t + \gamma q(s_{t+1}, a_{t+1})$	После каждого шага	On-policy: оценивает текущую политику поведения
Q-learning	$r_t + \gamma \max_a q(s_{t+1}, a)$	После каждого шага	Off-policy: оценивает жадную целевую политику

15.7. Q-обучение

Q-обучение (Q-learning) использует жадную оценку следующего действия:

$$q(s_t, a_t) \leftarrow q(s_t, a_t) + \alpha(r_t + \gamma \max_a q(s_{t+1}, a) - q(s_t, a_t)). \quad 15.20$$

Это метод вне целевой политики (off-policy method): политика поведения, по которой агент собирает опыт, может быть исследующей, например ε -жадной (ε -greedy), а целевая политика в обновлении использует жадный максимум по действиям.

Для исследования пространства действий часто применяют:

$$a_t = \begin{cases} \text{случайное действие} & \text{с вероятностью } \varepsilon \\ \arg \max_a q(s_t, a) & \text{с вероятностью } 1 - \varepsilon. \end{cases} \quad 15.21$$

Так агент балансирует исследование (exploration) и использование уже найденного поведения (exploitation).

15.8. Глубокое Q-обучение

Если пространство состояний большое, таблица $q(s, a)$ невозможна. Тогда используют нейронную сеть:

$$q_\phi(s, a) \approx q(s, a). \quad 15.22$$

Целевое значение:

$$y_t = r_t + \gamma \max_a q_{\phi^-}(s_{t+1}, a), \quad 15.23$$

где ϕ^- — параметры целевой сети (target network). Функция потерь:

$$L(\phi) = (y_t - q_\phi(s_t, a_t))^2. \quad 15.24$$

Градиентный шаг:

$$\phi \leftarrow \phi + \alpha (y_t - q_\phi(s_t, a_t)) \frac{\partial q_\phi(s_t, a_t)}{\partial \phi}. \quad 15.25$$

Глубокое Q-обучение связывает обучение с подкреплением с глубоким обучением: нейросеть заменяет таблицу ценностей и обобщает опыт между похожими состояниями.

Для устойчивости DQN использует две инженерные идеи.

1. **Буфер опыта** (experience replay). Переходы (s_t, a_t, r_t, s_{t+1}) сохраняют в буфере и обучают сеть на случайных мини-батчах из этого буфера. Это уменьшает корреляцию соседних переходов.
2. **Целевая сеть** (target network). Параметры ϕ^- обновляют редко, например копированием $\phi^- \leftarrow \phi$ раз в несколько тысяч шагов. Это делает целевое значение y_t более стабильным.

Если s_{t+1} является терминальным состоянием, будущая ценность отсутствует:

$$y_t = r_t. \quad 15.26$$

Иначе используют обычную цель

$$y_t = r_t + \gamma \max_a q_{\phi^-}(s_{t+1}, a). \quad 15.27$$

15.9. Градиент политики

В методах градиента политики (policy gradient methods) политика задается напрямую:

$$\pi_{\theta}(a \mid s). \quad 15.28$$

Вероятность траектории:

$$P(\tau \mid \theta) = P(s_1) \prod_{t=1}^T \pi_{\theta}(a_t \mid s_t) P(s_{t+1} \mid s_t, a_t). \quad 15.29$$

Цель:

$$J(\theta) = \mathbb{E}_{\tau \sim P(\tau \mid \theta)}[r(\tau)]. \quad 15.30$$

Обновление строится по градиенту:

$$\theta \leftarrow \theta + \alpha \nabla_{\theta} J(\theta). \quad 15.31$$

Используя трюк логарифмической производной (log-derivative trick):

$$\nabla_{\theta} J(\theta) = \mathbb{E}_{\tau}[r(\tau) \nabla_{\theta} \log P(\tau \mid \theta)]. \quad 15.32$$

Так как среда не зависит от θ , остается сумма градиентов логарифмов политики:

$$\nabla_{\theta} J(\theta) = \mathbb{E}_{\tau} \left[r(\tau) \sum_{t=1}^T \nabla_{\theta} \log \pi_{\theta}(a_t \mid s_t) \right]. \quad 15.33$$

Градиент политики удобен для непрерывных действий и стохастических политик, но требует контроля дисперсии, например базового уровня (baseline) или функций преимущества (advantage functions).

Базовый уровень $b(s_t)$ можно вычитать из отдачи без смещения математического ожидания градиента:

$$\nabla_{\theta} J(\theta) = \mathbb{E} \left[\sum_{t=1}^T (G_t - b(s_t)) \nabla_{\theta} \log \pi_{\theta}(a_t \mid s_t) \right]. \quad 15.34$$

Частый выбор - функция ценности состояния $b(s_t) = v_{\pi}(s_t)$. Тогда величина

$$A_{\pi}(s_t, a_t) = q_{\pi}(s_t, a_t) - v_{\pi}(s_t) \quad 15.35$$

называется преимуществом (advantage) и показывает, насколько выбранное действие лучше среднего действия политики в этом состоянии.

15.10. Итог

1. Обучение с подкреплением изучает последовательное принятие решений, где действие влияет на будущие состояния и награды.
2. MDP задает состояния, действия, переходы, награды и дисконтирование.
3. Функции v_π и q_π оценивают ожидаемую будущую награду.
4. Уравнения Беллмана связывают текущую ценность с наградой и ценностью следующего состояния.
5. Динамическое программирование требует известной модели среды.
6. Методы временных разностей обновляют ценности по одному шагу и используют TD-ошибку.
7. Q-обучение оценивает оптимальную функцию действия как off-policy метод.
8. Глубокое Q-обучение аппроксимирует $q(s, a)$ нейронной сетью.
9. Градиент политики оптимизирует параметры политики напрямую.

15.11. Проверочные вопросы

1. Чем обучение с подкреплением отличается от обучения с учителем?
2. Что входит в определение MDP?
3. Запишите дисконтированную отдачу G_t .
4. Чем $v_\pi(s)$ отличается от $q_\pi(s, a)$?
5. Запишите уравнение Беллмана для v_π .
6. В чем различие SARSA и Q-learning?
7. Что такое TD-ошибка?
8. Зачем в глубоком Q-обучении используется целевая сеть?
9. Как ϵ -жадная политика балансирует исследование и использование уже найденного поведения?
10. Выведите идею градиента политики через $\nabla_\theta \log \pi_\theta(a | s)$.

Источники

1. Prince S. J. D. *Understanding Deep Learning*. Chapter 19, Reinforcement learning. <https://udlbook.github.io/udlbook/>; notebooks: <https://github.com/udlbook/udlbook>.
2. Zhang A., Lipton Z. C., Li M., Smola A. J. *Dive into Deep Learning*. Chapter 17, Reinforcement Learning. <https://github.com/d2l-ai/d2l-en>.
3. Dive into Deep Learning. Reinforcement Learning. https://d2l.ai/chapter_reinforcement-learning/index.html.

Лекция 16. Диффузионные модели

Диффузионные модели являются вероятностными генеративными моделями: они обучаются обращать постепенное зашумление данных и затем генерируют новые объекты из шума. В лекции рассматриваются прямой диффузионный процесс, обратный процесс, вариационная нижняя оценка, переход к задаче предсказания шума и связь формул с практическим обучением нейронной сети.

16.1. Постановка проблемы

Пусть есть обучающая выборка изображений, сигналов или других объектов:

$$\mathbf{x} \sim p_{\text{data}}(\mathbf{x}). \quad 16.1$$

Нужно построить модель, которая может генерировать новые образцы из распределения, близкого к p_{data} . Диффузионный подход вводит цепочку латентных переменных

$$\mathbf{z}_1, \mathbf{z}_2, \dots, \mathbf{z}_T, \quad 16.2$$

где прямой процесс постепенно добавляет шум к данным, а обратный процесс учится этот шум удалять.

Идея:

$$\mathbf{x} \rightarrow \mathbf{z}_1 \rightarrow \dots \rightarrow \mathbf{z}_T \approx \mathcal{N}(\mathbf{0}, \mathbf{I}), \quad 16.3$$

а генерация идет в обратную сторону:

$$\mathbf{z}_T \rightarrow \mathbf{z}_{T-1} \rightarrow \dots \rightarrow \mathbf{z}_1 \rightarrow \mathbf{x}. \quad 16.4$$

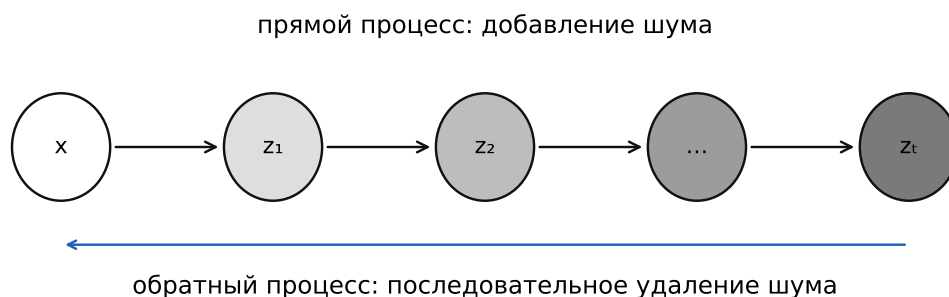


Рис. 16.1. Прямая диффузия добавляет шум, а обратная цепочка учится его удалять.

16.2. Прямой процесс зашумления

Прямой процесс задается фиксированным расписанием шумов β_t :

$$\mathbf{z}_1 = \sqrt{1 - \beta_1} \mathbf{x} + \sqrt{\beta_1} \epsilon_1, \quad 16.5$$

$$\mathbf{z}_t = \sqrt{1 - \beta_t} \mathbf{z}_{t-1} + \sqrt{\beta_t} \epsilon_t, \quad t = 2, \dots, T. \quad 16.6$$

Здесь

$$\epsilon_t \sim \mathcal{N}(\mathbf{0}, \mathbf{I}). \quad 16.7$$

В вероятностной форме:

$$q(\mathbf{z}_1 | \mathbf{x}) = \mathcal{N}(\sqrt{1 - \beta_1} \mathbf{x}, \beta_1 \mathbf{I}), \quad 16.8$$

$$q(\mathbf{z}_t | \mathbf{z}_{t-1}) = \mathcal{N}(\sqrt{1 - \beta_t} \mathbf{z}_{t-1}, \beta_t \mathbf{I}). \quad 16.9$$

Совместное распределение:

$$q(\mathbf{z}_{1:T} | \mathbf{x}) = q(\mathbf{z}_1 | \mathbf{x}) \prod_{t=2}^T q(\mathbf{z}_t | \mathbf{z}_{t-1}). \quad 16.10$$

16.3. Закрытая форма для $\mathbf{z}_t$

Обозначим одношаговый коэффициент сохранения сигнала и его накопленное произведение:

$$\alpha_t = 1 - \beta_t, \quad |(\alpha)_t = \prod_{\tau=1}^t \alpha_\tau = \prod_{\tau=1}^t (1 - \beta_\tau). \quad 16.11$$

Тогда зашумленный объект на шаге t можно получить напрямую:

$$\mathbf{z}_t = \sqrt{|(\alpha)_t|} \mathbf{x} + \sqrt{1 - |(\alpha)_t|} \epsilon, \quad 16.12$$

где $\epsilon \sim \mathcal{N}(\mathbf{0}, \mathbf{I})$. Поэтому

$$q(\mathbf{z}_t | \mathbf{x}) = \mathcal{N}(\sqrt{|(\alpha)_t|} \mathbf{x}, (1 - |(\alpha)_t|) \mathbf{I}). \quad 16.13$$

Эта формула важна практически: во время обучения можно выбрать случайный шаг t , сразу построить $\mathbf{z}_t$ и обучать сеть предсказывать шум или исходный объект.

16.4. Обратный процесс

Если прямой процесс известен, то идеальный обратный шаг имел бы вид

$$q(\mathbf{z}_{t-1} \mid \mathbf{z}_t). \quad 16.14$$

Но он зависит от неизвестного распределения данных. Поэтому вводится обучаемая модель со средним, которое предсказывает нейронная сеть:

$$p_\phi(\mathbf{z}_{t-1} \mid \mathbf{z}_t) = \mathcal{N}(\boldsymbol{\mu}_\phi(\mathbf{z}_t, t), \sigma_t^2 \mathbf{I}). \quad 16.15$$

Начальное распределение генерации:

$$p(\mathbf{z}_T) = \mathcal{N}(\mathbf{0}, \mathbf{I}). \quad 16.16$$

Совместная модель:

$$p_\phi(\mathbf{x}, \mathbf{z}_{1:T}) = p_\phi(\mathbf{x} \mid \mathbf{z}_1) \prod_{t=2}^T p_\phi(\mathbf{z}_{t-1} \mid \mathbf{z}_t) p(\mathbf{z}_T). \quad 16.17$$

Задача обучения — подобрать параметры ϕ , чтобы обратные переходы приближали истинные условные распределения.

16.5. Условный обратный шаг и цель обучения

Если известен исходный объект $\mathbf{x}$, то можно выписать

$$q(\mathbf{z}_{t-1} \mid \mathbf{z}_t, \mathbf{x}) = \mathcal{N}(\boldsymbol{\mu}_q(\mathbf{z}_t, \mathbf{x}, t), \tilde{\beta}_t \mathbf{I}), \quad 16.18$$

где среднее в формуле UDL имеет вид

$$\boldsymbol{\mu}_q = \frac{1 - |(\alpha)_{t-1}|}{1 - |(\alpha)_t|} \sqrt{\alpha_t} \mathbf{z}_t + \frac{\sqrt{|(\alpha)_{t-1}|} \beta_t}{1 - |(\alpha)_t|} \mathbf{x}. \quad 16.19$$

Тогда естественная цель — сделать предсказанное среднее $\boldsymbol{\mu}_\phi(\mathbf{z}_t, t)$ близким к $\boldsymbol{\mu}_q$:

$$\| \boldsymbol{\mu}_q(\mathbf{z}_t, \mathbf{x}, t) - \boldsymbol{\mu}_\phi(\mathbf{z}_t, t) \|^2. \quad 16.20$$

Эта цель возникает из дивергенции Кульбака-Лейблера (Kullback-Leibler divergence, KL-дивергенция) между двумя нормальными распределениями в вариационной нижней оценке.

16.6. Вариационная нижняя оценка для диффузионной модели

Правдоподобие данных содержит интеграл по латентной траектории:

$$p_\phi(\mathbf{x}) = \int p_\phi(\mathbf{x}, \mathbf{z}_{1:T}) d\mathbf{z}_{1:T}. \quad 16.21$$

Напрямую оптимизировать его трудно. Поэтому используют вариационную нижнюю оценку (Evidence Lower Bound, ELBO):

$$\text{ELBO}[\phi] = \int q(\mathbf{z}_{1:T} | \mathbf{x}) \log \left[\frac{p_\phi(\mathbf{x}, \mathbf{z}_{1:T})}{q(\mathbf{z}_{1:T} | \mathbf{x})} \right] d\mathbf{z}_{1:T}. \quad 16.22$$

После преобразований она распадается на реконструкционный член, KL-дивергенцию последнего зашумленного состояния к стандартному нормальному распределению и сумму KL-дивергенций по обратным шагам:

$$\text{ELBO} \approx \mathbb{E}_{q(\mathbf{z}_1 | \mathbf{x})} [\log p_\phi(\mathbf{x} | \mathbf{z}_1)] - D_{KL}(q(\mathbf{z}_T | \mathbf{x}) \parallel p(\mathbf{z}_T)) - \sum_{t=2}^T \mathbb{E}_{q(\mathbf{z}_t | \mathbf{x})} [D_{KL}(q(\mathbf{z}_{t-1} | \mathbf{z}_t, \mathbf{x}) \parallel p_\phi(\mathbf{z}_{t-1} | \mathbf{z}_t))]. \quad 16.23$$

В учебных выводах последний prior term часто опускают или считают малым, если $q(\mathbf{z}_T | \mathbf{x})$ близко к $\mathcal{N}(\mathbf{0}, \mathbf{I})$.

Максимизация ELBO эквивалентна минимизации ошибки обратных шагов.

16.7. Переход к предсказанию шума

Из формулы прямого процесса:

$$\mathbf{z}_t = \sqrt{|\alpha)_t} \mathbf{x} + \sqrt{1 - |\alpha)_t} \epsilon. \quad 16.24$$

Можно выразить исходный объект:

$$\mathbf{x} = \frac{1}{\sqrt{|\alpha)_t}} \mathbf{z}_t - \frac{\sqrt{1 - |\alpha)_t}}{\sqrt{|\alpha)_t}} \epsilon. \quad 16.25$$

Если параметризовать обратный шаг через сеть, предсказывающую шум,

$$\epsilon_\phi(\mathbf{z}_t, t) \approx \epsilon, \quad 16.26$$

то практическая функция потерь принимает простой вид:

$$L(\phi) = \mathbb{E}_{\mathbf{x}, t, \epsilon} \left[\left\| \epsilon_{\phi} \left(\sqrt{|\alpha|_t} \mathbf{x} + \sqrt{1 - |\alpha|_t} \epsilon, t \right) - \epsilon \right\|^2 \right]. \quad 16.27$$

Это центральная формула обучения моделей, подобных вероятностным диффузионным моделям удаления шума (Denoising Diffusion Probabilistic Models, DDPM): сеть получает зашумленный объект и номер шага, а учится восстанавливать добавленный шум.

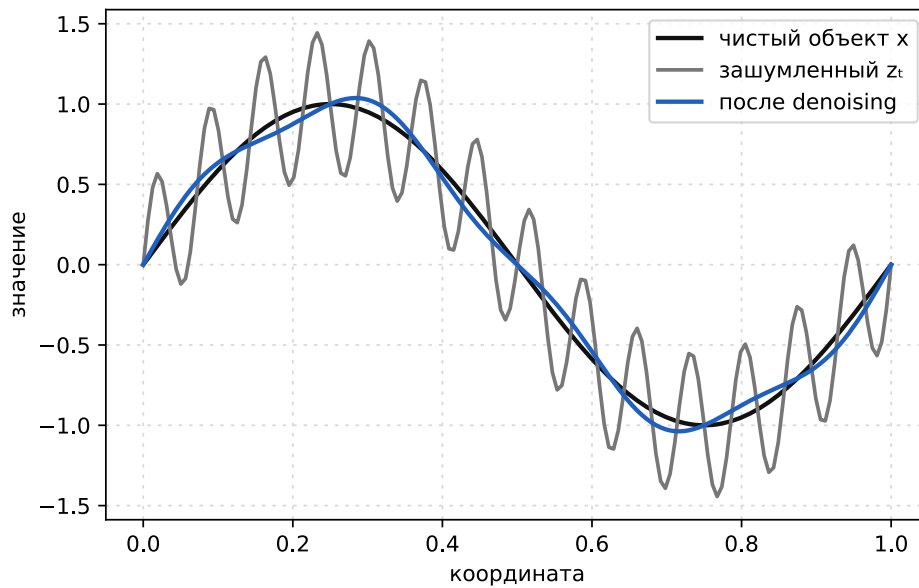


Рис. 16.2. Сеть удаления шума приближает чистый объект через предсказание добавленного шума.

16.8. Алгоритм обучения

Один шаг обучения:

1. выбрать объект $\mathbf{x}$ из обучающей выборки;
2. выбрать шаг $t \sim \text{Uniform}\{1, \dots, T\}$;
3. выбрать шум $\epsilon \sim \mathcal{N}(\mathbf{0}, \mathbf{I})$;
4. построить

$$\mathbf{z}_t = \sqrt{|\alpha|_t} \mathbf{x} + \sqrt{1 - |\alpha|_t} \epsilon; \quad 16.28$$

5. сделать градиентный шаг по

$$\left\| \epsilon_{\phi}(\mathbf{z}_t, t) - \epsilon \right\|^2. \quad 16.29$$

На практике ϵ_ϕ часто реализуют U-Net-подобной сверточной архитектурой, потому что она хорошо сочетает локальные признаки и многомасштабный контекст.

16.9. Алгоритм генерации

Генерация начинается с шума:

$$\mathbf{z}_T \sim \mathcal{N}(\mathbf{0}, \mathbf{I}). \quad 16.30$$

Затем для $t = T, T - 1, \dots, 1$ модель предсказывает шум или среднее обратного перехода и делает шаг удаления шума (denoising):

$$\mathbf{z}_{t-1} \sim p_\phi(\mathbf{z}_{t-1} \mid \mathbf{z}_t). \quad 16.31$$

После последнего шага получается выборка $\hat{\mathbf{x}}$. Генерация обычно медленнее, чем один прямой проход (forward pass) генеративно-состязательной сети (Generative Adversarial Network, GAN) или вариационного автоэнкодера (Variational Autoencoder, VAE), потому что требует много обратных шагов, но качество и устойчивость обучения часто высоки.

16.10. Условная генерация

Если нужно генерировать объект по условию c , например по классу или текстовому описанию, модель заменяют на

$$\epsilon_\phi(\mathbf{z}_t, t, c). \quad 16.32$$

Идея наведения классификатором (classifier guidance) состоит в том, чтобы усилить направление, повышающее вероятность условия по отдельному классификатору. В общем виде обратный шаг может корректироваться градиентом:

$$\mathbf{z}_{t-1} = \hat{\mathbf{z}}_{t-1} + \sigma_t^2 \nabla_{\mathbf{z}_t} \log P(c \mid \mathbf{z}_t) + \sigma_t \epsilon. \quad 16.33$$

В наведении без классификатора (classifier-free guidance) одна и та же сеть обучается с условием и без него, а затем смешивают два предсказания шума:

$$\hat{\epsilon}_\phi(\mathbf{z}_t, t, c) = (1 + \omega) \epsilon_\phi(\mathbf{z}_t, t, c) - \omega \epsilon_\phi(\mathbf{z}_t, t, \emptyset), \quad 16.34$$

где $\omega \geq 0$ задает силу наведения.

Так вероятностная модель связывается с управляемой генерацией.

16.11. Итог

1. Диффузионная модель задает прямой процесс зашумления и обучаемый обратный процесс удаления шума.
2. Прямой процесс имеет закрытую форму $\mathbf{z}_t = \sqrt{|\alpha)_t} \mathbf{x} + \sqrt{1 - |\alpha)_t} \epsilon$.
3. Обратные переходы моделируются нормальными распределениями с нейросетевым средним.
4. ELBO приводит к сумме реконструкционного члена и KL-дивергенций между истинными и обучаемыми обратными шагами.
5. Практическое обучение часто сводится к задаче среднеквадратичной ошибки (Mean Squared Error, MSE) для предсказания шума.
6. Генерация начинается с гауссовского шума (Gaussian noise) и постепенно удаляет шум.
7. Условная генерация добавляет информацию о классе, тексте или другом условии в сеть удаления шума.

16.12. Проверочные вопросы

1. Какую задачу решает диффузионная генеративная модель?
2. Запишите прямой процесс $q(\mathbf{z}_t | \mathbf{z}_{t-1})$.
3. Чем отличаются одношаговый коэффициент $\alpha_t = 1 - \beta_t$ и накопленный коэффициент $|\alpha)_t$?
4. Выведите закрытую форму $\mathbf{z}_t = \sqrt{|\alpha)_t} \mathbf{x} + \sqrt{1 - |\alpha)_t} \epsilon$.
5. Почему обратный процесс нужно обучать?
6. Как ELBO связана с KL-дивергенциями обратных переходов?
7. Почему задачу обучения удобно переписать как предсказание шума?
8. Запишите практическую функцию потерь MSE для ϵ_ϕ .
9. Почему генерация в диффузионных моделях обычно итеративна?
10. Как условие c входит в условную диффузионную модель?

Источники

1. Prince S. J. D. *Understanding Deep Learning*. Chapter 18, Diffusion models. <https://udlbook.github.io/udlbook/>; notebooks: <https://github.com/udlbook/udlbook>.
2. Ho J., Jain A., Abbeel P. Denoising Diffusion Probabilistic Models. arXiv:2006.11239. <https://arxiv.org/abs/2006.11239>.

3. Song Y., Sohl-Dickstein J., Kingma D. P., Kumar A., Ermon S., Poole B. Score-Based Generative Modeling through Stochastic Differential Equations. arXiv:2011.13456. <https://arxiv.org/abs/2011.13456>.

Лекция 17. Байесовы нейронные сети

Байесовы нейронные сети (Bayesian Neural Networks, BNN) рассматривают параметры модели как случайные величины и позволяют описывать неопределенность предсказаний. В лекции рассматриваются апостериорное распределение параметров, апостериорное предсказательное распределение (posterior predictive distribution), вариационная аппроксимация, вариационная нижняя оценка (Evidence Lower Bound, ELBO), связь с регуляризацией и практические методы приближенного байесовского вывода.

17.1. Постановка проблемы

Обычная нейронная сеть после обучения дает одну точечную оценку параметров:

$$\hat{\phi} = \arg \min_{\phi} L(\phi). \quad 17.1$$

Такой подход часто дает хорошую точность, но плохо отделяет уверенное предсказание от неуверенного. В задачах медицины, автономного управления, финансов и научного моделирования важно понимать, насколько модель уверена в ответе.

Байесовский подход задает априорное распределение (prior) на параметры:

$$p(\phi), \quad 17.2$$

и после наблюдения данных $\mathcal{D}$ строит апостериорное распределение (posterior):

$$p(\phi \mid \mathcal{D}) = \frac{p(\mathcal{D} \mid \phi)p(\phi)}{p(\mathcal{D})}. \quad 17.3$$

Предсказание усредняет ответы по возможным параметрам:

$$p(y \mid \mathbf{x}, \mathcal{D}) = \int p(y \mid \mathbf{x}, \phi)p(\phi \mid \mathcal{D})d\phi. \quad 17.4$$

Именно этот интеграл отличает байесову нейронную сеть от обычной.

17.2. Правдоподобие, априорное и апостериорное распределения

Для обучения с учителем (supervised learning) данные имеют вид

$$\mathcal{D} = \{(\mathbf{x}_i, y_i)\}_{i=1}^N. \quad 17.5$$

Если наблюдения условно независимы при фиксированных параметрах:

$$p(\mathcal{D} | \phi) = \prod_{i=1}^N p(y_i | \mathbf{x}_i, \phi). \quad 17.6$$

Апостериорное распределение:

$$p(\phi | \mathcal{D}) = \frac{\prod_{i=1}^N p(y_i | \mathbf{x}_i, \phi) p(\phi)}{\int \prod_{i=1}^N p(y_i | \mathbf{x}_i, \phi) p(\phi) d\phi}. \quad 17.7$$

Знаменатель

$$p(\mathcal{D}) = \int p(\mathcal{D} | \phi) p(\phi) d\phi \quad 17.8$$

называется свидетельством, или маргинальным правдоподобием (evidence). Для глубоких сетей этот интеграл практически невычислим, поэтому нужны приближения.

17.3. Оценка максимума апостериорной вероятности и связь с регуляризацией

Оценка максимума апостериорной вероятности (Maximum A Posteriori, MAP):

$$\hat{\phi}_{\text{MAP}} = \arg \max_{\phi} \log p(\mathcal{D} | \phi) + \log p(\phi). \quad 17.9$$

Если априорное распределение нормально:

$$p(\phi) = \mathcal{N}(\mathbf{0}, \sigma_p^2 \mathbf{I}), \quad 17.10$$

то

$$\log p(\phi) = C - \frac{1}{2\sigma_p^2} \|\phi\|_2^2. \quad 17.11$$

Максимизация апостериорного распределения эквивалентна минимизации

$$L_{\text{MAP}}(\phi) = - \sum_{i=1}^N \log p(y_i | \mathbf{x}_i, \phi) + \lambda \|\phi\|_2^2. \quad 17.12$$

Здесь λ пропорционален $\frac{1}{2\sigma_p^2}$ с учетом выбранной нормировки функции потерь. Так регуляризация с нормой L_2 (L2-регуляризация) получает байесовскую интерпретацию: она соответствует нормальному априорному распределению на веса. Но MAP все равно дает точечную оценку и не хранит полное апостериорное распределение.

17.4. Апостериорное предсказательное распределение

Главное предсказание байесовой модели:

$$p(y | \mathbf{x}, \mathcal{D}) = \int p(y | \mathbf{x}, \phi) p(\phi | \mathcal{D}) d\phi. \quad 17.13$$

Если есть выборки параметров $\phi^{(1)}, \dots, \phi^{(M)}$ из апостериорного распределения, интеграл приближается средним:

$$p(y | \mathbf{x}, \mathcal{D}) \approx \frac{1}{M} \sum_{m=1}^M p(y | \mathbf{x}, \phi^{(m)}). \quad 17.14$$

Для регрессии можно оценить среднее и дисперсию:

$$\mathbb{E}[y | \mathbf{x}, \mathcal{D}] \approx \frac{1}{M} \sum_{m=1}^M f_{\phi^{(m)}}(\mathbf{x}), \quad 17.15$$

$$|(f)(\mathbf{x})| = \frac{1}{M} \sum_{m=1}^M f_{\phi^{(m)}}(\mathbf{x}). \quad 17.16$$

$$\text{Var}[y | \mathbf{x}, \mathcal{D}] \approx \frac{1}{M} \sum_{m=1}^M (f_{\phi^{(m)}}(\mathbf{x}) - |(f)(\mathbf{x})|)^2 + \sigma_{\text{noise}}^2. \quad 17.17$$

Первый член отражает неопределенность параметров, второй — шум наблюдений.

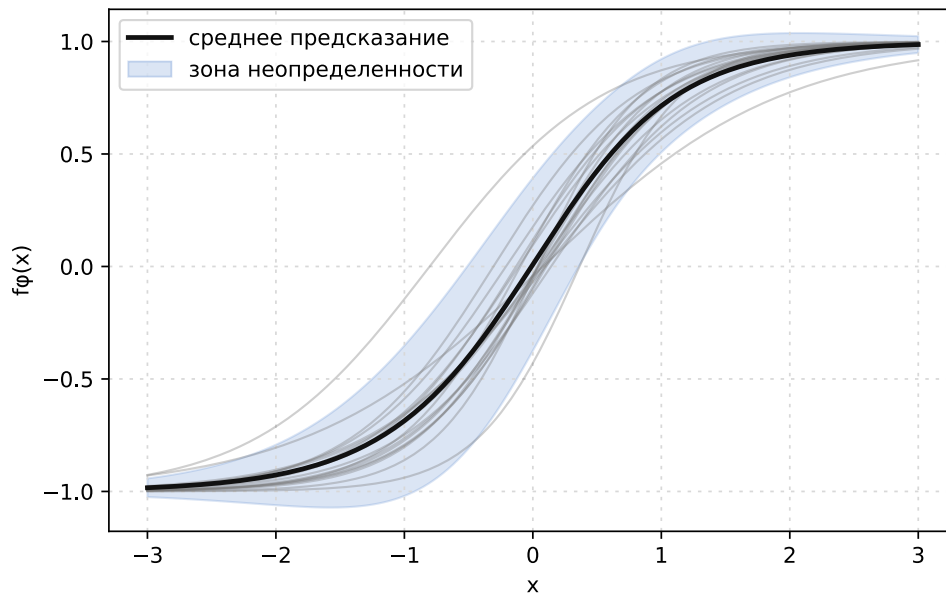


Рис. 17.1. Апостериорное предсказательное распределение усредняет ответы по разным значениям параметров.

17.5. Эпистемическая и алеаторическая неопределенность

В байесовых нейронных сетях обычно различают два вида неопределенности.

Эпистемическая неопределенность связана с нехваткой данных. Она уменьшается, когда модель видит больше релевантных наблюдений. В формулах выше ей соответствует дисперсия предсказаний по выборкам параметров из $p(\phi \mid \mathcal{D})$.

Алеаторическая неопределенность связана с шумом самой задачи. Даже при бесконечном количестве данных наблюдение может оставаться неоднозначным. В регрессионной записи ей соответствует член σ_{noise}^2 , а в общем случае она задается условным распределением $p(y \mid \mathbf{x}, \phi)$.

Для принятия решений важно не только получить класс или число, но и оценить риск:

$$\hat{y} = \arg \max_y p(y \mid \mathbf{x}, \mathcal{D}), \quad u(\mathbf{x}) = H[p(y \mid \mathbf{x}, \mathcal{D})], \quad 17.18$$

где H — энтропия предсказательного распределения.

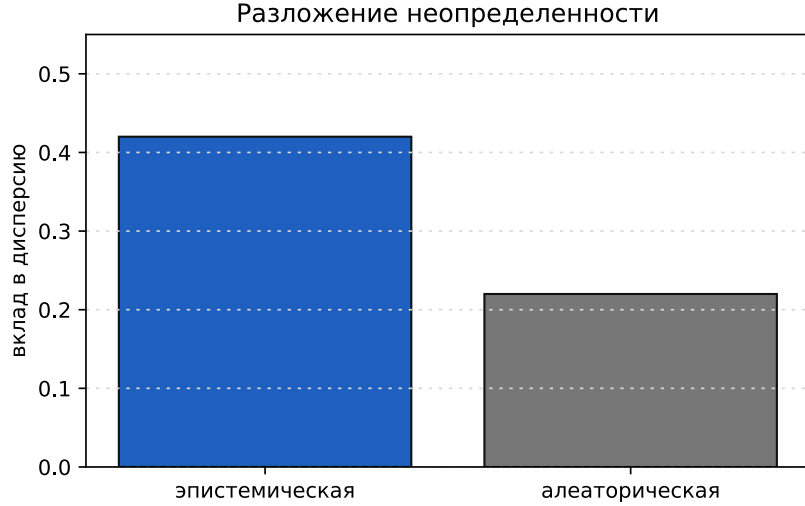


Рис. 17.2. Эпистемическая и алеаторическая части вносят разные вклады в общую неопределенность.

17.6. Вариационный вывод

Точное апостериорное распределение $p(\phi \mid \mathcal{D})$ недоступно. Вариационный вывод вводит аппроксимацию

$$q_{\theta}(\phi) \approx p(\phi \mid \mathcal{D}), \quad 17.19$$

и минимизирует

$$D_{KL}(q_{\theta}(\phi) \parallel p(\phi \mid \mathcal{D})). \quad 17.20$$

Эквивалентно максимизируется вариационная нижняя оценка (ELBO):

$$\text{ELBO}(\theta) = \mathbb{E}_{q_{\theta}(\phi)}[\log p(\mathcal{D} \mid \phi)] - D_{KL}(q_{\theta}(\phi) \parallel p(\phi)). \quad 17.21$$

Первый член требует хорошо объяснять данные, второй удерживает приближение рядом с априорным распределением. Отрицательная ELBO становится функцией потерь:

$$L(\theta) = -\mathbb{E}_{q_{\theta}(\phi)}[\log p(\mathcal{D} \mid \phi)] + D_{KL}(q_{\theta}(\phi) \parallel p(\phi)). \quad 17.22$$

Частый выбор — факторизованное нормальное распределение:

$$q_{\theta}(\phi) = \prod_k \mathcal{N}(\phi_k \mid \mu_k, \sigma_k^2). \quad 17.23$$

Тогда обучаются параметры μ_k и σ_k .

17.7. Трюк репараметризации

Чтобы оценивать градиенты по параметрам распределения, используют трюк репараметризации (reparameterization trick):

$$\phi_k = \mu_k + \sigma_k \epsilon_k, \quad \epsilon_k \sim \mathcal{N}(0, 1). \quad 17.24$$

Тогда случайность вынесена в ϵ_k , а μ_k, σ_k остаются дифференцируемыми параметрами. Оценка первого члена ELBO:

$$\mathbb{E}_{q_\theta(\phi)}[\log p(\mathcal{D} \mid \phi)] \approx \frac{1}{M} \sum_{m=1}^M \log p(\mathcal{D} \mid \phi^{(m)}). \quad 17.25$$

Так байесова нейросеть обучается теми же оптимизаторами, что и обычная, но прямой проход (forward pass) включает сэмплирование весов.

17.8. Практические приближения

Полноценное вариационное апостериорное распределение для всех весов дорого. Поэтому используют приближения.

Метод Монте-Карло со случайным выключением нейронов (Monte Carlo dropout, MC dropout) интерпретирует случайное выключение нейронов (dropout) на этапе вывода как сэмплирование подмоделей. Несколько прямых проходов дают ансамбль предсказаний:

$$p(y \mid \mathbf{x}, \mathcal{D}) \approx \frac{1}{M} \sum_{m=1}^M p(y \mid \mathbf{x}, \phi, \mathbf{m}^{(m)}), \quad 17.26$$

где $\mathbf{m}^{(m)}$ — случайная маска выключения нейронов.

Глубокие ансамбли (deep ensembles) обучают несколько независимых сетей:

$$p(y \mid \mathbf{x}) \approx \frac{1}{M} \sum_{m=1}^M p(y \mid \mathbf{x}, \phi_m). \quad 17.27$$

Это не строгое апостериорное распределение, но часто дает сильную практическую оценку неопределенности.

Аппроксимация Лапласа (Laplace approximation) аппроксимирует апостериорное распределение около MAP-решения нормальным распределением:

$$p(\phi \mid \mathcal{D}) \approx \mathcal{N}(\hat{\phi}_{\text{MAP}}, \mathbf{H}^{-1}), \quad 17.28$$

где $\mathbf{H}$ — гессиан отрицательного логарифма апостериорного распределения (log posterior).

Таблица 17.1. Практические приближения байесовского вывода.

Метод	Идея	Сильная сторона	Ограничение
Вариационный вывод	Обучить распределение $q_{\theta}(\phi)$	Явная вероятностная постановка	Дорого для больших сетей и зависит от семейства q_{θ}
MC dropout	Сэмплировать маски dropout на выводе	Простая реализация поверх обычной сети	Приближение грубое и зависит от схемы dropout
Глубокие ансамбли	Обучить несколько независимых сетей	Часто сильная практическая неопределенность	Требует несколько обучений модели
Аппроксимация Лапласа	Нормальное приближение около MAP	Связывает кривизну с неопределенностью	Трудно хранить полный гессиан

Качество неопределенности проверяют отдельно от точности. Для классификации используют калибровку: среди объектов, которым модель назначила вероятность около 0.8, правильных ответов должно быть примерно 80%. Для одного объекта полезны энтропия предсказания

$$H[p(y \mid \mathbf{x}, \mathcal{D})] = - \sum_y p(y \mid \mathbf{x}, \mathcal{D}) \log p(y \mid \mathbf{x}, \mathcal{D}) \quad 17.29$$

и взаимная информация между параметрами и предсказанием, которая выделяет эпистемическую часть неопределенности. Обычная softmax-вероятность точечной сети часто переоценивает уверенность на объектах вне обучающего распределения.

17.9. Когда нужны байесовы нейронные сети

Байесовы нейронные сети (BNN) особенно полезны, когда ошибка уверенной модели стоит дорого:

1. медицинская диагностика;

2. активное обучение;
3. обнаружение данных вне обучающего распределения (out-of-distribution data);
4. управление и робототехника;
5. научное моделирование с малым числом наблюдений.

Если модель видит объект, сильно отличный от обучающих данных, хорошее апостериорное предсказательное распределение должно стать более неопределенным. Это свойство нельзя гарантировать одной только softmax-вероятностью обычной сети.

17.10. Итог

1. Байесова нейронная сеть рассматривает параметры как случайные величины.
2. Апостериорное распределение $p(\phi \mid \mathcal{D})$ объединяет правдоподобие (likelihood) и априорное распределение.
3. MAP-оценка связана с регуляризацией, но остается точечной.
4. Предсказание BNN усредняет $p(y \mid x, \phi)$ по апостериорному распределению параметров.
5. Эпистемическая неопределенность связана с нехваткой данных, алеаторическая — с шумом задачи.
6. Вариационный вывод заменяет недоступное апостериорное распределение приближением $q_{\theta}(\phi)$.
7. ELBO содержит ожидаемое логарифмическое правдоподобие (log-likelihood) и KL-регуляризацию к априорному распределению.
8. Трюк репараметризации позволяет обучать вариационные параметры градиентными методами.
9. Метод Монте-Карло со случайным выключением нейронов, глубокие ансамбли и аппроксимация Лапласа являются практическими приближениями байесовского вывода.
10. Неопределенность нужно проверять через калибровку, энтропию предсказания и поведение на объектах вне обучающего распределения.

17.11. Проверочные вопросы

1. Чем байесова нейронная сеть отличается от обычной сети с точечными весами?

2. Запишите формулу апостериорного распределения $p(\phi \mid \mathcal{D})$.
3. Как апостериорное предсказательное распределение использует неопределенность параметров?
4. Почему L2-регуляризация связана с нормальным априорным распределением?
5. Чем эпистемическая неопределенность отличается от алеаторической?
6. Что минимизирует вариационный вывод?
7. Запишите ELBO для вариационного апостериорного распределения $q_{\theta}(\phi)$.
8. Как работает трюк репараметризации для нормального распределения весов?
9. Почему метод Монте-Карло со случайным выключением нейронов можно использовать для приближенной оценки неопределенности?
10. В каких прикладных задачах оценка неопределенности особенно важна?

Источники

1. Prince S. J. D. *Understanding Deep Learning*. Chapter 9, Bayesian approach; Chapter 17, Variational autoencoders and latent variable inference. <https://udlbook.github.io/udlbook/>; notebooks: <https://github.com/udlbook/udlbook>.
2. RBC Borealis. Bayesian Neural Networks. <https://rbcborealis.com/research-blogs/bayesian-neural-networks/>.
3. Zhang A., Lipton Z. C., Li M., Smola A. J. *Dive into Deep Learning*. Appendix on probability/statistics and Gaussian processes. <https://github.com/d2l-ai/d2l-en>.
4. Gal Y., Ghahramani Z. Dropout as a Bayesian Approximation: Representing Model Uncertainty in Deep Learning. arXiv:1506.02142. <https://arxiv.org/abs/1506.02142>.

Приложение А. Теорема Колмогорова о представимости функций

Исторический контекст

Тринадцатая проблема Гильберта ставила вопрос о том, можно ли выразить корень уравнения

$$x^7 + ax^3 + bx^2 + cx + 1 = 0 \quad 90.1$$

как функцию коэффициентов через суперпозиции непрерывных функций двух переменных. Работы А. Н. Колмогорова и В. И. Арнольда показали более сильный результат: непрерывные функции многих переменных можно представлять через суперпозиции непрерывных функций одного переменного и операции сложения.

Формулировка теоремы

Теорема Колмогорова (О представимости непрерывных функций). Для любого целого $n \geq 2$ существуют непрерывные функции

$$\psi^{pq} : [0, 1] \rightarrow \mathbb{R}, \quad p = 1, \dots, n, \quad q = 1, \dots, 2n + 1, \quad 90.2$$

такие, что для всякой непрерывной функции

$$f : [0, 1]^n \rightarrow \mathbb{R} \quad 90.3$$

существуют непрерывные функции $\chi_q : \mathbb{R} \rightarrow \mathbb{R}$, для которых выполнено представление

$$f(x_1, \dots, x_n) = \sum_{q=1}^{2n+1} \chi_q \left(\sum_{p=1}^n \psi^{pq}(x_p) \right). \quad 90.4$$

При этом функции ψ^{pq} не зависят от выбора функции f .

Доказательство для случая $n = 2$

Общий случай доказывается аналогично, поэтому достаточно рассмотреть двумерный куб $[0, 1]^2$. Тогда нужно получить представление

$$f(x_1, x_2) = \sum_{q=1}^5 \chi_q (\psi_q(x_1) + \phi_q(x_2)), \quad 90.5$$

где функции ψ_q и ϕ_q строятся заранее, а функции χ_q подбираются уже под конкретную непрерывную функцию f .

1. Покрывание единичного квадрата

На отрезке $[0, 1]$ строится семейство мелких сегментов $A_{qi}^{(k)}$, зависящих от номера шага k , индекса i и номера семейства $q \in \{1, \dots, 5\}$. Затем рассматриваются кубики

$$S_{qi}^{(k)} = A_{qi}^{(k)} \times A_{qi}^{(k)}. \quad 90.6$$

Сегменты выбираются так, чтобы при фиксированном k и разных q соответствующие системы были немного сдвинуты друг относительно друга. Это приводит к ключевому свойству.

Лемма 1. Для каждого фиксированного k система всех кубиков $S_{qi}^{(k)}$ покрывает квадрат $[0, 1]^2$ так, что каждая точка покрывается не менее трех раз.

Идея доказательства состоит в том, что для каждой точки оценивается сумма характеристических функций покрывающих сегментов. Из-за выбранных сдвигов среди пяти семейств по крайней мере три обязательно содержат данную точку.

2. Построение внутренних координат

Далее вводятся вспомогательные константы и интервалы, которые позволяют каждому сегменту сопоставить числовой образ. На этой основе строятся непрерывные монотонно возрастающие функции ψ_q и ϕ_q на $[0, 1]$.

Лемма 2. Можно выбрать параметры построения так, чтобы:

1. образы пересекающихся сегментов согласовывались;
2. образы различных сегментов при фиксированных k и q не пересекались;
3. полученные функции ψ_q и ϕ_q были непрерывными и однозначно определенными на всем отрезке $[0, 1]$.

Из этих свойств следует, что суммы $\psi_q(x_1) + \phi_q(x_2)$ различают достаточно большое число кубиков и позволяют кодировать положение точки (x_1, x_2) .

3. Построение функций χ_q

После фиксации функций ψ_q и ϕ_q строятся функции χ_q . Это делается итерационно. На шаге r вводится приближение

$$f_r(x_1, x_2) = \sum_{q=1}^5 \chi_q^{(r)} (\psi_q(x_1) + \phi_q(x_2)) \quad 90.7$$

и остаток

$$M_r = \sup_{(x_1, x_2) \in [0, 1]^2} |f(x_1, x_2) - f_r(x_1, x_2)|. \quad 90.8$$

Благодаря лемме о тройном покрытии на каждом шаге можно выбрать добавки к функциям $\chi_q^{(r)}$, уменьшающие остаток. В результате получается оценка

$$M_r \leq \frac{5}{6} M_{r-1}. \quad 90.9$$

Следовательно, погрешность убывает геометрически.

4. Сходимость и завершение доказательства

Разности между соседними приближениями функций $\chi_q^{(r)}$ оцениваются сверху членами сходящегося ряда вида

$$\sum_{r=1}^{\infty} \frac{1}{3} M_{r-1}. \quad 90.10$$

По признаку Вейерштрасса ряды, определяющие функции χ_q , сходятся равномерно, а потому их пределы непрерывны. Переходя к пределу при $r \rightarrow \infty$, получаем

$$f(x_1, x_2) = \sum_{q=1}^5 \chi_q (\psi_q(x_1) + \phi_q(x_2)). \quad 90.11$$

Тем самым теорема доказана для $n = 2$, а общий случай устанавливается тем же методом.

Что важно для нейронных сетей

Теорема Колмогорова показывает, что многомерную непрерывную зависимость можно разложить в суперпозицию одномерных непрерывных функций и сложения. В теории нейронных сетей это служит одним из ранних математических оснований идеи, что сложные функции можно собирать из простых нелинейных блоков.

Источники

1. Колмогоров А. Н. О представлении непрерывных функций нескольких переменных в виде суперпозиции непрерывных функций одного переменного. Доклады АН СССР, 1957, т. 114, № 5, с. 953-956.
2. Арнольд В. И. О представлении функций нескольких переменных в виде суперпозиции функций меньшего числа переменных. Математическое просвещение.
3. Legasy-источник репозитория: `legasy`-источник курса.

Приложение Б. Фундаментальные теоремы нейровычислений

Введение

В этом приложении собраны фундаментальные результаты, которые лежат в основе теоретического понимания аппроксимирующих возможностей нейронных сетей.

Теорема Стоуна-Вейерштрасса

Пусть X — компактное пространство, а $C(X)$ — пространство всех вещественных непрерывных функций на X . Рассмотрим подмножество $B \subseteq C(X)$, для которого выполнены условия:

1. если $f, g \in B$, то $fg \in B$ и $\alpha f + \beta g \in B$ для любых вещественных α, β ;
2. постоянная функция 1 принадлежит B ;
3. предел любой равномерно сходящейся последовательности функций из B также принадлежит B .

Теорема 3.1 (Стоуна-Вейерштрасса). Тогда $B = C(X)$ тогда и только тогда, когда множество B разделяет точки пространства X , то есть для любых двух различных точек $s_1, s_2 \in X$ существует функция $x \in B$, такая что

$$x(s_1) \neq x(s_2). \quad 91.1$$

Смысл теоремы. Если класс функций замкнут относительно линейных операций, умножения и равномерного предельного перехода, а также умеет различать точки пространства, то этого класса достаточно для приближения любой непрерывной функции на компакте.

Обобщенная теорема Стоуна

Пусть X — компактное пространство, $C(X)$ — пространство непрерывных функций на X с вещественными значениями, $C(\mathbb{R})$ — пространство непрерывных функций на действительной оси, а $E \subseteq C(X)$ — замкнутое линейное подпространство. Предположим, что:

1. $1 \in E$;
2. функции из E разделяют точки пространства X ;
3. пространство E замкнуто относительно нелинейной унарной операции $f \in C(\mathbb{R})$.

Теорема 3.2 (Обобщенная теорема Стоуна). При этих условиях

$$E = C(X). \quad 91.2$$

Интерпретация. Для аппроксимации непрерывных функций достаточно не только линейных операций и умножения, но и произвольной фиксированной нелинейной унарной операции. Это важный мост к нейросетям, где нелинейность задается функцией активации.

Леммы, на которых строится доказательство

Для доказательства обобщенной теоремы вводится множество

$$P = \{p \in C(\mathbb{R}) \mid p(E) \subseteq E\}, \quad 91.3$$

то есть множество всех непрерывных функций одной переменной, которые сохраняют пространство E при суперпозиции.

У этого множества выделяются четыре ключевых свойства:

1. P — полугруппа относительно суперпозиции функций.
2. P — замкнутое линейное подпространство в $C(\mathbb{R})$ в топологии равномерной сходимости на компактах.
3. P содержит константу 1 и тождественную функцию $\text{id}(x) = x$.
4. P содержит хотя бы одну непрерывную нелинейную функцию.

Далее используются три леммы.

Лемма 3.1. В множестве P существует дважды непрерывно дифференцируемая функция, не являющаяся линейной.

Идея доказательства: нелинейную функцию сглаживают оператором осреднения. Благодаря замкнутости множества P полученная функция остается в P , но становится более регулярной.

Лемма 3.2. Если в P существует дважды непрерывно дифференцируемая нелинейная функция, то функция

$$q(x) = x^2 \quad 91.4$$

принадлежит P .

Идея доказательства: из нелинейности выбирается точка x_0 , в которой вторая производная не равна нулю, после чего строится нормированная разность

$$r(x) = \frac{2g(x + x_0) - 2g(x_0) - xg'(x_0)}{g''(x_0)}, \quad 91.5$$

и затем используется предельный переход

$$\lim_{\varepsilon \rightarrow 0} \frac{r(\varepsilon x)}{\varepsilon^2} = x^2. \quad 91.6$$

Так как P замкнуто, отсюда следует, что $x^2 \in P$.

Лемма 3.3. Если функция x^2 принадлежит P , то множество P является кольцом: для любых $f, g \in P$ их произведение также принадлежит P .

Доказательство основано на тождестве

$$fg = \frac{1}{2} \left((f + g)^2 - f^2 - g^2 \right). \quad 91.7$$

Поскольку P замкнуто относительно суперпозиции и линейных операций, отсюда следует $fg \in P$.

Итог доказательства обобщенной теоремы

Из лемм 3.1-3.3 следует, что множество P содержит все многочлены, так как оно содержит 1 , $\text{id}(x) = x$, функцию x^2 и замкнуто относительно умножения и линейных комбинаций. Тогда по классической теореме Вейерштрасса множество P содержит все непрерывные функции одной переменной на компактах. Следовательно, пространство E замкнуто относительно достаточно богатого класса суперпозиций, и это приводит к равенству

$$E = C(X). \quad 91.8$$

Следствие для нейровычислений

Следствие. С помощью операции суперпозиции и линейных комбинаций из любых нелинейных элементов можно получить любую функцию многих переменных с любой заданной точностью.

Именно это утверждение делает теоремы Стоуна и их обобщения фундаментальными для нейровычислений: нейронная сеть строит

сложную функцию из линейных преобразований и повторяющихся нелинейных активаций. Теорема объясняет, почему такая архитектура в принципе может аппроксимировать широкий класс непрерывных зависимостей.

Источники

1. Stone M. H. The generalized Weierstrass approximation theorem. *Mathematics Magazine*, 1948, Vol. 21, pp. 167-183, 237-254.
2. Cybenko G. Approximation by superposition of a sigmoidal function. *Mathematics of Control, Signals, and Systems*, 1989, Vol. 2, pp. 303-314.
3. Горбань А. Н., Россиев Д. А. *Нейронные сети на персональном компьютере*. Новосибирск: Наука, 1996.
4. Иосида К. *Функциональный анализ*. Москва: Мир, 1967.
5. Legacy-источник репозитория: legacy-источник курса.

Приложение В. Вычисление производных

Введение

В этом приложении собраны векторные приемы быстрого вычисления градиентов нейронных сетей [1]. Главная идея состоит в том, чтобы не вычислять производную по одному отдельному параметру, если можно сохранить все выражения в матрично-векторной форме. Такой подход ближе к реальной реализации обратного распространения ошибки и позволяет сразу видеть размерности всех промежуточных величин.

Векторные градиенты и матрица Якоби

При ручном выводе полезно уметь находить производную нейронной сети по одному элементу матрицы весов. Но на практике это медленно и плохо показывает структуру вычислений. Поэтому градиенты обычно записывают в компактном матрично-векторном виде. Базовым блоком такого вывода является матрица Якоби.

Пусть задана функция $f : \mathbb{R}^n \rightarrow \mathbb{R}^m$, которая переводит вектор длины n в вектор длины m :

$$f(x) = [f_1(x_1, \dots, x_n), f_2(x_1, \dots, x_n), \dots, f_m(x_1, \dots, x_n)]. \quad 92.1$$

Ее якобиан имеет размер $m \times n$:

$$\frac{\partial f}{\partial x} = \begin{bmatrix} \frac{\partial f_1}{\partial x_1} & \dots & \frac{\partial f_1}{\partial x_n} \\ \vdots & \ddots & \vdots \\ \frac{\partial f_m}{\partial x_1} & \dots & \frac{\partial f_m}{\partial x_n} \end{bmatrix}. \quad 92.2$$

Иначе говоря, элемент якобиана с индексами (i, j) равен обычной частной производной:

$$\left(\frac{\partial f}{\partial x} \right)_{ij} = \frac{\partial f_i}{\partial x_j}. \quad 92.3$$

Якобиан удобен тем, что правило дифференцирования сложной функции для векторных отображений превращается в произведение якобианов.

Рассмотрим простой пример. Пусть функция $f(x) = [f_1(x), f_2(x)]$ переводит скаляр в двумерный вектор, а функция $g(y) = [g_1(y_1, y_2), g_2(y_1, y_2)]$ переводит двумерный вектор в двумерный вектор. Их композиция имеет вид

$$g(f(x)) = [g_1(f_1(x), f_2(x)), g_2(f_1(x), f_2(x))]. \quad 92.4$$

По обычному цепному правилу производная композиции равна

$$\frac{\partial g}{\partial x} = \begin{bmatrix} \frac{\partial}{\partial x} g_1(f_1(x), f_2(x)) \\ \frac{\partial}{\partial x} g_2(f_1(x), f_2(x)) \end{bmatrix} = \begin{bmatrix} \frac{\partial g_1}{\partial f_1} \frac{\partial f_1}{\partial x} + \frac{\partial g_1}{\partial f_2} \frac{\partial f_2}{\partial x} \\ \frac{\partial g_2}{\partial f_1} \frac{\partial f_1}{\partial x} + \frac{\partial g_2}{\partial f_2} \frac{\partial f_2}{\partial x} \end{bmatrix}. \quad 92.5$$

Та же запись получается как произведение двух якобианов:

$$\frac{\partial g}{\partial x} = \frac{\partial g}{\partial f} \frac{\partial f}{\partial x} = \begin{bmatrix} \frac{\partial g_1}{\partial f_1} & \frac{\partial g_1}{\partial f_2} \\ \frac{\partial g_2}{\partial f_1} & \frac{\partial g_2}{\partial f_2} \end{bmatrix} \begin{bmatrix} \frac{\partial f_1}{\partial x} \\ \frac{\partial f_2}{\partial x} \end{bmatrix}. \quad 92.6$$

Полезные тождества

Ниже перечислены тождества для простых операций. Их достаточно, чтобы быстро выводить градиенты многих нейронных сетей. В каждом пункте важно следить не только за формулой, но и за размерностями.

Тождество 1. Матрица, умноженная на вектор-столбец, по вектору-столбцу. Пусть

$$z = Wx, \quad W \in \mathbb{R}^{n \times m}, \quad x \in \mathbb{R}^m. \quad 92.7$$

Тогда z можно рассматривать как функцию, которая переводит m -мерный вектор в n -мерный. Ее якобиан имеет размер $n \times m$. Компонента z_i равна

$$z_i = \sum_{k=1}^m W_{ik} x_k. \quad 92.8$$

Поэтому элемент якобиана равен

$$\left(\frac{\partial z}{\partial x} \right)_{ij} = \frac{\partial z_i}{\partial x_j} = \frac{\partial}{\partial x_j} \sum_{k=1}^m W_{ik} x_k = W_{ij}. \quad 92.9$$

Следовательно,

$$\frac{\partial z}{\partial x} = W. \quad 92.10$$

Тождество 2. Вектор-строка, умноженная на матрицу, по вектору-строке. Если

$$z = xW, \quad 92.11$$

то аналогичный вывод дает

$$\frac{\partial z}{\partial x} = W^T. \quad 92.12$$

Тождество 3. Вектор по самому себе. Если $z = x$, то $z_i = x_i$, а значит

$$\left(\frac{\partial z}{\partial x}\right)_{ij} = \frac{\partial z_i}{\partial x_j} = \begin{cases} 1 & \text{при } i = j \\ 0 & \text{иначе} \end{cases}. \quad 92.13$$

Значит, якобиан является единичной матрицей:

$$\frac{\partial z}{\partial x} = I. \quad 92.14$$

При применении цепного правила такой множитель исчезает, потому что умножение на единичную матрицу не меняет матрицу или вектор.

Тождество 4. Поэлементная функция от вектора. Пусть $z = \varphi(x)$, где функция φ применяется к вектору поэлементно. Тогда $z_i = \varphi(x_i)$, и

$$\left(\frac{\partial z}{\partial x}\right)_{ij} = \begin{cases} \varphi'(x_i) & \text{при } i = j \\ 0 & \text{иначе} \end{cases}. \quad 92.15$$

Поэтому якобиан является диагональной матрицей:

$$\frac{\partial z}{\partial x} = \text{diag}(\varphi'(x)). \quad 92.16$$

Умножение на диагональную матрицу эквивалентно поэлементному умножению на ее диагональ. Поэтому при применении цепного правила такую операцию часто записывают через $\odot \varphi'(x)$.

Тождество 5. Матрица, умноженная на вектор-столбец, по матрице. Пусть

$$z = Wx, \quad \delta = \frac{\partial J}{\partial z}. \quad 92.17$$

Нужно найти

$$\frac{\partial J}{\partial W} = \frac{\partial J}{\partial z} \frac{\partial z}{\partial W} = \delta \frac{\partial z}{\partial W}. \quad 92.18$$

Здесь есть тонкость. Если J - скалярная функция потерь, а $W \in \mathbb{R}^{n \times m}$, то формально J можно рассматривать как функцию от nm входов - эле-

ментов матрицы W . Якобиан $\frac{\partial J}{\partial W}$ тогда был бы строкой размера $1 \times nm$. Для оптимизации это неудобно: практическое соглашение состоит в том, чтобы хранить градиент в матрице той же формы, что и параметр:

$$\frac{\partial J}{\partial W} = \begin{bmatrix} \frac{\partial J}{\partial W_{11}} & \cdots & \frac{\partial J}{\partial W_{1m}} \\ \vdots & \ddots & \vdots \\ \frac{\partial J}{\partial W_{n1}} & \cdots & \frac{\partial J}{\partial W_{nm}} \end{bmatrix}. \quad 92.19$$

Чтобы избежать тензора $\frac{\partial J}{\partial W}$, рассмотрим один элемент W_{ij} . Для компоненты

$$z_k = \sum_{l=1}^m W_{kl} x_l \quad 92.20$$

получаем

$$\frac{\partial z_k}{\partial W_{ij}} = \sum_{l=1}^m x_l \frac{\partial W_{kl}}{\partial W_{ij}}. \quad 92.21$$

Производная $\frac{\partial W_{kl}}{\partial W_{ij}}$ равна 1 только при $i = k$ и $j = l$, иначе она равна 0. Поэтому

$$\frac{\partial z_k}{\partial W_{ij}} = \begin{cases} x_j & \text{при } k = i \\ 0 & \text{иначе} \end{cases}. \quad 92.22$$

Теперь по цепному правилу

$$\frac{\partial J}{\partial W_{ij}} = \frac{\partial J}{\partial z} \frac{\partial z}{\partial W_{ij}} = \sum_{k=1}^n \delta_k \frac{\partial z_k}{\partial W_{ij}} = \delta_i x_j. \quad 92.23$$

Следовательно, матрица градиента имеет элементы $\delta_i x_j$, то есть

$$\frac{\partial J}{\partial W} = \delta^\top x^\top. \quad 92.24$$

Если δ и x уже записаны как столбцы, та же формула читается как внешнее произведение $\delta x^\top$; в этом приложении мы сохраняем ориентации из якобианной записи.

Тождество 6. Вектор-строка, умноженная на матрицу, по матрице. Если

$$z = xW, \quad \delta = \frac{\partial J}{\partial z}, \quad 92.25$$

то аналогичное вычисление дает

$$\frac{\partial J}{\partial W} = x^\top \delta. \quad 92.26$$

Тождество 7. Кросс-энтропия по логитам. Пусть

$$\hat{y} = \text{softmax}(\theta), \quad J = \text{CE}(y, \hat{y}). \quad 92.27$$

Тогда

$$\frac{\partial J}{\partial \theta} = \hat{y} - y. \quad 92.28$$

Если y задан как вектор-столбец, то в якобианной записи это же выражение может появляться как $(\hat{y} - y)^\top$.

Эти тождества позволяют быстро получать градиенты для многих сетевых блоков. Для тренировки полезно самостоятельно вывести якобианы для скалярного произведения двух векторов, поэлементного произведения двух векторов и евклидовой нормы вектора. Иногда тождества удобно запомнить, а иногда достаточно проверить размерности: например, только один порядок произведения δ и x даст матрицу той же формы, что и W , если W не квадратная.

Соглашение о форме градиента

Якобианная формулировка удобна для цепного правила: достаточно перемножать якобианы. Но при стохастическом градиентном спуске удобнее использовать соглашение «форма градиента совпадает с формой параметра». Тогда обновление параметра записывается напрямую:

$$W \leftarrow W - \eta \frac{\partial J}{\partial W}. \quad 92.29$$

По этой причине итоговый ответ для параметра-столбца обычно транспонируют так, чтобы градиент тоже был столбцом. Можно идти другим путем и с самого начала поддерживать это соглашение о форме градиента. Тогда отдельные якобианные тождества могут выглядеть иначе, но размерности производных все равно должны согласовываться с параметрами.

Пример: однослойная нейронная сеть

Рассмотрим пример полного вывода градиентов для однослойной нейронной сети с кросс-энтропийной функцией потерь. Прямой проход задается так:

$$\begin{aligned}
x &= \text{вход}, \\
z &= Wx + b_1, \\
h &= \text{ReLU}(z), \\
\theta &= Uh + b_2, \\
\hat{y} &= \text{softmax}(\theta), \\
J &= \text{CE}(y, \hat{y}).
\end{aligned} \tag{92.30}$$

Полезно разбивать модель на максимально простые части. Поэтому здесь отдельно введены z и θ : они отделяют линейные преобразования от нелинейных функций активации.

Размерности параметров:

$$x \in \mathbb{R}^{D_x \times 1}, \quad b_1 \in \mathbb{R}^{D_h \times 1}, \quad W \in \mathbb{R}^{D_h \times D_x}, \quad b_2 \in \mathbb{R}^{N_c \times 1}, \quad U \in \mathbb{R}^{N_c \times D_h}. \tag{92.31}$$

Здесь D_x - размер входа, D_h - размер скрытого слоя, а N_c - число классов. Нужно получить все градиенты сети:

$$\frac{\partial J}{\partial U}, \quad \frac{\partial J}{\partial b_2}, \quad \frac{\partial J}{\partial W}, \quad \frac{\partial J}{\partial b_1}, \quad \frac{\partial J}{\partial x}. \tag{92.32}$$

Вспомним, что $\text{ReLU}(x) = \max(x, 0)$. Ее производная равна 1 при $x > 0$ и 0 иначе. Эту производную можно записывать через результат самой активации:

$$\text{ReLU}'(x) = \text{sgn}(\text{ReLU}(x)). \tag{92.33}$$

Сначала запишем цепное правило для U и b_2 :

$$\frac{\partial J}{\partial U} = \frac{\partial J}{\partial \hat{y}} \frac{\partial \hat{y}}{\partial \theta} \frac{\partial \theta}{\partial U}, \quad \frac{\partial J}{\partial b_2} = \frac{\partial J}{\partial \hat{y}} \frac{\partial \hat{y}}{\partial \theta} \frac{\partial \theta}{\partial b_2}. \tag{92.34}$$

В обеих формулах повторяется произведение $\frac{\partial J}{\partial \hat{y}} \hat{y}$ и $\frac{\partial \hat{y}}{\partial \theta} \theta$, то есть $\frac{\partial J}{\partial \theta}$. Чтобы не писать и не вычислять его несколько раз, введем промежуточные ошибки:

$$\delta_1 = \frac{\partial J}{\partial \theta}, \quad \delta_2 = \frac{\partial J}{\partial z}. \tag{92.35}$$

Их можно понимать как сигналы ошибки, которые передаются назад к θ и z в алгоритме обратного распространения. Первый сигнал сразу следует из тождества для softmax и кросс-энтропии:

$$\delta_1 = \frac{\partial J}{\partial \theta} = (\hat{y} - y)^\top. \tag{92.36}$$

Второй сигнал получаем цепным правилом:

$$\delta_2 = \frac{\partial J}{\partial z} = \frac{\partial J}{\partial \theta} \frac{\partial \theta}{\partial h} \frac{\partial h}{\partial z} = \delta_1 U \odot \text{ReLU}'(z) = \delta_1 U \odot \text{sgn}(h). \quad 92.37$$

Проверка размерностей подтверждает согласованность вывода:

$$\underbrace{\delta_2}_{1 \times D_h} = \underbrace{\delta_1}_{1 \times N_c} \underbrace{U}_{N_c \times D_h} \odot \underbrace{\text{sgn}(h)}_{D_h}. \quad 92.38$$

Теперь используем ошибки δ_1 и δ_2 для вычисления градиентов. Для параметров-столбцов итог транспонируется, чтобы соблюсти соглашение о совпадении формы градиента и параметра:

$$\frac{\partial J}{\partial U} = \delta_1 \frac{\partial \theta}{\partial U} = \delta_1^\top h^\top, \quad 92.39$$

$$\frac{\partial J}{\partial b_2} = \delta_1 \frac{\partial \theta}{\partial b_2} = \delta_1^\top, \quad 92.40$$

$$\frac{\partial J}{\partial W} = \delta_2 \frac{\partial z}{\partial W} = \delta_2^\top x^\top, \quad 92.41$$

$$\frac{\partial J}{\partial b_1} = \delta_2 \frac{\partial z}{\partial b_1} = \delta_2^\top, \quad 92.42$$

$$\frac{\partial J}{\partial x} = (\delta_2 W)^\top. \quad 92.43$$

Полученный набор формул показывает общий шаблон обратного прохода: сначала вычисляются ошибки на выходных линейных переменных, затем из них строятся градиенты матриц весов как внешние произведения ошибки и входа соответствующего слоя, а градиенты смещений совпадают с ошибками слоя с нужной ориентацией.

Источники

1. Clark K. *Computing Neural Network Gradients*. Stanford CS224n notes, 2019.