Вычислительная линейная алгебра

Если метод Ньютона из главы 2 был одним маленьким уравнением, упрямо повторяющимся, то линейная алгебра — это язык, на котором говорят сразу о тысячах уравнений и миллионах чисел. Этой главой мы покажем, что многие фокусы современного искусственного интеллекта — от того, как нейросеть «узнаёт» лицо человека, до того, как поисковик ранжирует страницы Интернета, — держатся всего на нескольких разложениях матриц. Мы посмотрим четыре истории: лица, котиков, Интернет и устойчивость нейросетей.

Зачем линейной алгебре отдельная глава

Вы уже встречались с матрицами в школьном курсе — как с прямоугольными таблицами чисел. В современном ИИ матрица — это **универсальный носитель данных**.

- Чёрно-белая фотография 1024×1024 — это матрица размера 1024×1024, где каждое число — яркость пикселя от 0 (чёрный) до 255 (белый). Цветная — три такие матрицы.
- Текст из N слов словаря — это вектор частот длины N (модель «мешка слов»), и набор из M документов — матрица $M \times N$.
- Социальная сеть из N пользователей — матрица $N \times N$, где элемент (i, j) равен 1, если i подписан на j.
- Веса одного слоя нейросети — матрица W, через которую входной сигнал умножается, чтобы получить выходной.

С такими матрицами надо уметь *что-то делать*: сжимать, сравнивать, упорядочивать, обновлять. И почти всегда оказывается, что нужная операция выражается через одно и то же фундаментальное разложение матрицы — **сингулярное**. Мы изучим его и применим к четырём задачам:

1. **Узнать лицо** по фотографии — *eigenfaces* (§0.5).
2. **Сжать пространство признаков** — *eigencats* и теорема Эккарта–Янга (§0.4, §0.6).
3. **Упорядочить весь Интернет** — *PageRank* (§0.7).
4. **Понять, когда нейросеть устойчива** — *оценки Липшицевости* (§0.8).

Прежде чем перейти к сюжетам, разберёмся с одной *очень* земной проблемой: с тем, как компьютер хранит числа.

$AB \neq BA$, или почему линейная алгебра не похожа на школьную

В школе вы привыкли, что от перестановки множителей произведение не меняется: $2 \cdot 3 = 3 \cdot 2$. Для квадратных матриц это *в общем случае не так* — они не коммутируют.

Самая наглядная демонстрация — повороты трёхмерного предмета. Возьмите смартфон, поверните его сначала на 90° вокруг горизонтальной оси X (вершина <<нырнула>> вперёд), потом на 90° вокруг вертикальной оси Y (предмет повернулся вправо). Теперь повторите эксперимент, но в обратном порядке. Финальные положения смартфона будут *разными*. Каждый поворот задаётся ортогональной матрицей 3×3, и эти матрицы не коммутируют:

$$R_y(90°)\,R_x(90°) \neq R_x(90°)\,R_y(90°). \tag{1}$$

Геймеры одной из лучших игр 2023 года *The Legend of Zelda: Tears of the Kingdom* ощущают этот эффект каждый раз, когда крутят объект в инвентаре. Подробный разбор сюжета: t.me/fminxyz/15.

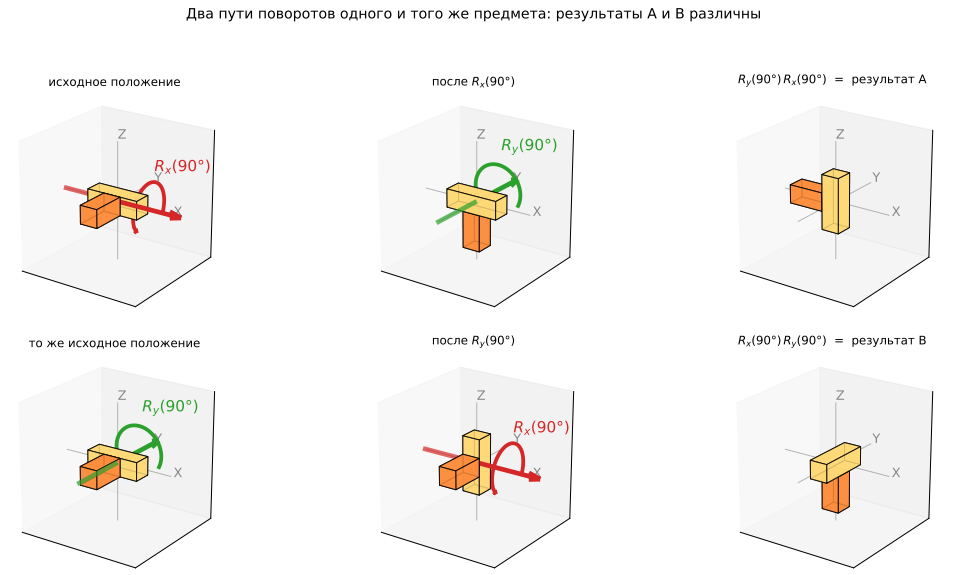

Рисунок 1. image

На двумерной плоскости повороты, кстати, коммутируют: $R(\alpha)R(\beta) = R(\beta)R(\alpha) = R(\alpha+\beta)$. Так что эффект чисто <<трёхмерный>>: чем больше степеней свободы, тем менее послушна алгебра.

Когда арифметика лжёт: числа с плавающей точкой

Откройте Python и наберите:

```
    (
 sequence(
   styled(child: [>>>], ..),
   styled(child: [ ], ..),
   styled(child: [0.1], ..),
   styled(child: [ ], ..),
   styled(child: [+], ..),
   styled(child: [ ], ..),
   styled(child: [0.2], ..),
 ),
 styled(child: [0.30000000000000004], ..),
 sequence(
   styled(child: [>>>], ..),
   styled(child: [ ], ..),
   styled(child: [0.1], ..),
   styled(child: [ ], ..),
   styled(child: [+], ..),
   styled(child: [ ], ..),
   styled(child: [0.2], ..),
   styled(child: [ ], ..),
   styled(child: [==], ..),
   styled(child: [ ], ..),
   styled(child: [0.3], ..),
 ),
 styled(child: [False], ..),
)
```

Это не баг Python. То же самое выдаст любой современный язык — C, C++, Java, JavaScript. И этому есть фундаментальная причина: **двоичная система счисления** не умеет точно представить десятичную дробь $0,1$.

Двоичные дроби и почему $0,1$ *— это «*$0,333\ldots$*»*

В десятичной системе $1/3 = 0,3333\ldots$ — бесконечная дробь. То же самое в двоичной системе случается с $0,1$:

$$0,1_{10} = 0,0001\,1001\,1001\,1001\ldots_2 \tag{2}$$

Дробь периодическая. Любой реальный компьютер хранит её, оборвав после конечного числа разрядов (обычно 52 двоичных знака после запятой — стандарт *IEEE 754, double precision*). Поэтому число, которое программист записывает как 0.1, на самом деле равно

$$0,1 \approx 0,100\,000\,000\,000\,000\,005\,551\,115\ldots \tag{3}$$

Когда мы складываем такое «искажённое $0,1$» с искажённым $0,2$, ошибки усиливаются — и получается тот самый «лишний хвостик» $4 \cdot 10^{-17}$, который мы видели в листинге.

> **i** Машинная точность $\varepsilon_{\text{маш}}$
>
> **Машинной точностью** называют наименьшее положительное число ε, такое что в компьютерной арифметике $1 + \varepsilon \neq 1$. Для стандарта double оно равно $\varepsilon_{\text{маш}} = 2^{-52} \approx 2,22 \cdot 10^{-16}$.
>
> Это означает: какое бы хорошее число x вы ни записали, оно известно компьютеру лишь с относительной точностью $\sim \varepsilon_{\text{маш}}$.

Когда это становится опасным: катастрофическое сокращение

Маленькая ошибка в 17-м знаке — мелочь? Не всегда. Рассмотрим безобидное на вид вычисление:

$$g(x) = \frac{1 - \cos x}{x^2} \qquad \text{при} \quad x = 10^{-8}. \tag{4}$$

По формуле Тейлора $1 - \cos x \approx x^2/2$, так что $g(10^{-8}) \approx 0,5$. Что выдаст компьютер?

```
  (
 sequence(
   styled(child: [>>>], ..),
   styled(child: [ ], ..),
   styled(child: [from], ..),
   styled(child: [ math ], ..),
   styled(child: [import], ..),
   styled(child: [ cos], ..),
 ),
 sequence(
   styled(child: [>>>], ..),
   styled(child: [ x ], ..),
   styled(child: [=], ..),
   styled(child: [ ], ..),
   styled(child: [1e-8], ..),
 ),
 sequence(
   styled(child: [>>>], ..),
   styled(child: [ (], ..),
   styled(child: [1], ..),
   styled(child: [ ], ..),
   styled(child: [-], ..),
   styled(child: [ cos(x)) ], ..),
   styled(child: [/], ..),
```

```
    styled(child: [ x], ..),
    styled(child: [**], ..),
    styled(child: [2], ..),
  ),
  sequence(
    styled(child: [0.0], ..),
    styled(child: [          ], ..),
    styled(child: [# вместо 0.5!], ..),
  ),
)
```

Что произошло? Число $\cos(10^{-8}) \approx 1 - 5 \cdot 10^{-17}$ так близко к единице, что компьютер просто хранит его как *ровно единицу* (поправка $5 \cdot 10^{-17}$ меньше $\varepsilon_{\text{маш}}$). Разность $1 - \cos x$ обнулилась, и весь ответ обнулился вместе с ней.

Это явление называется **катастрофическим сокращением** (*catastrophic cancellation*): когда из двух близких чисел вычитают и теряют все значащие цифры разом.

Пример 0.1. Спасение через переписывание формулы

Воспользуемся тождеством $1 - \cos x = 2\sin^2(x/2)$:

$$g(x) = \frac{2\sin^2(x/2)}{x^2} = \left(\frac{\sin(x/2)}{x/2}\right)^2. \tag{5}$$

Внутри теперь нет вычитания близких чисел, и Python выдаёт 0.4999999999999983 — погрешность лишь в последнем знаке.

Важно

Главный практический вывод. Если ваш численный алгоритм выдаёт странный ответ, первая гипотеза, которую следует проверить, — это не «бага в библиотеке», а *катастрофическое сокращение* в вашей формуле. Часто его можно убрать, переписав формулу алгебраически эквивалентным, но численно устойчивым способом.

Историческая справка

Стандарт *IEEE 754* был принят в 1985 г., в значительной мере благодаря Уильяму Кэхэну (*William Kahan*, премия Тьюринга 1989 г.). До этого каждый производитель процессоров делал плавающую арифметику по-своему, и одна и та же программа на разных машинах могла давать заметно разные ответы. Кэхэн также является автором знаменитого **алгоритма компенсированного суммирования**, который позволяет складывать миллионы чисел с почти двойной точностью.

Сингулярное разложение SVD: главный фокус линейной алгебры

Геометрическая идея

Возьмём любую матрицу A размера $m \times n$ и применим её к единичному кругу в $\mathbb{R}^n$ (то есть к множеству всех векторов x с $\| x \| = 1$, где $\| \cdot \|$ — обычная евклидова длина). Что получится в $\mathbb{R}^m$?

⚠ Теорема 0.2. Сингулярное разложение (SVD)

Для любой вещественной матрицы A размера $m \times n$ существует разложение

$$A = U \Sigma \bar{V} \tag{6}$$

где

- U — *ортогональная* матрица $m \times m$ (поворот в $\mathbb{R}^m$);
- V — ортогональная матрица $n \times n$ (поворот в $\mathbb{R}^n$);
- Σ — «диагональная» матрица $m \times n$ с неотрицательными числами $\sigma_1 \geq \sigma_2 \geq \ldots \geq \sigma_r > 0$ на диагонали (и нулями ниже), где $r = \operatorname{rank} A$.

Числа σ_i называются **сингулярными числами** матрицы A.

Содержательно (0.1) говорит: *любая* линейная операция — это три простых действия подряд:

$$\underbrace{\bar{V}}_{\text{поворот в исходном пространстве}} \rightarrow \underbrace{\Sigma}_{\text{растяжение по координатным осям}} \rightarrow \underbrace{U}_{\text{поворот в новом пространстве}} \tag{7}$$

.

Геометрически это значит: образ единичной сферы под действием A — это **эллипсоид**, у которого полуоси равны $\sigma_1, \sigma_2, \ldots$ (см. рис. 0.1).

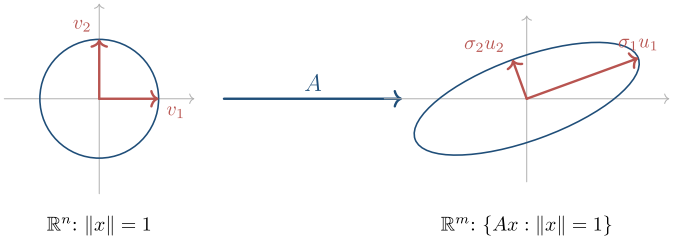

Рисунок 2. Рис. 0.1. SVD геометрически. Матрица A переводит единичный круг в эллипс, полуоси которого равны сингулярным числам σ_1, σ_2, а направления полуосей — столбцы матрицы U (левые сингулярные векторы). Векторы v_1, v_2 (столбцы V, правые сингулярные векторы) — это те направления в исходном пространстве, которые A переводит в полуоси эллипса.

Связь с собственными значениями

Сингулярные числа — это, по сути, переодетые собственные значения. А именно:

$$A^{\top}A=(U\Sigma V^{\top})^{\top}(U\Sigma V^{\top})=V\,\Sigma^{\top}\Sigma\,V^{\top}. \tag{8}$$

Матрица $\Sigma^{\top}\Sigma$ — диагональная, на её диагонали стоят $\sigma_1^2, \sigma_2^2, \ldots$. Значит, σ_i^2 — это *собственные значения* симметричной матрицы $A^{\top}A$, а столбцы V — её собственные векторы. Эта формула важна нам по двум причинам: (1) она обосновывает существование SVD (через известную спектральную теорему для симметричных матриц), (2) она будет рабочим инструментом в § 0.5, когда мы будем строить базис «собственных лиц» (e

> **Историческая справка**
>
> Сингулярное разложение независимо открыли итальянский математик *Эудженио Бельтрами* (1873 г.) и француз *Камилл Жордан* (1874 г.). В современном виде, как разложение прямоугольной матрицы, его сформулировал *Карл Эккарт* с *Гейлом Янгом* в 1936 г. Численно надёжный алгоритм вычисления SVD появился лишь в 1965 г. (Голуб и Кахан) — и стал одним из ключевых алгоритмов XX века. Сегодня он реализован в любой математической библиотеке: numpy.linalg.svd, scipy.linalg.svd, torch.svd.

Лучшее малоранговое приближение: теорема Эккарта–Янга

Самое замечательное свойство SVD состоит в том, что оно позволяет *ужимать* матрицу, выбрасывая всё «несущественное».

Разложение по слагаемым ранга один

Перепишем (0.1) по-другому. Обозначим столбцы матрицы U через $u_1, \ldots, u_m$ (это *левые сингулярные векторы*), а столбцы V — через $v_1, \ldots, v_n$ (*правые сингулярные векторы*). Тогда A есть сумма r слагаемых, каждое из которых имеет ранг один:

$$A=\sum_{i=1}^{r}\sigma_i\,u_i v_i^{\top}. \tag{9}$$

Каждое слагаемое $u_i v_i^{\top}$ — это «вертикальный вектор умножить на горизонтальный», и оно само по себе является матрицей ранга один (рис. 0.2).

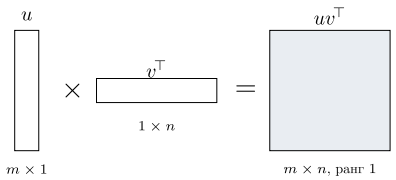

Рисунок 3. Рис. 0.2. Матрица ранга один: каждый столбец результата пропорционален u, каждая строка — пропорциональна $v^{\top}$. Чтобы её хранить, достаточно $m+n$ чисел вместо mn.

Если оборвать сумму (0.3) на k-м слагаемом ($k<r$), получим приближение:

$$A_k=\sum_{i=1}^{k}\sigma_i\,u_i v_i^{\top}. \tag{10}$$

Это матрица того же размера $m \times n$, но ранга всего лишь k.

Теорема Эккарта–Янга: A_k — наилучшее приближение

⚠ Теорема 0.3. Эккарт–Янг–Мирский (1936/1960)

Среди всех матриц B размера $m \times n$ ранга не более k матрица A_k из (0.4) даёт *наименьшую* ошибку в смысле спектральной нормы:

$$\min_{\operatorname{rank} B \le k} \| A - B \|_2 = \| A - A_k \|_2 = \sigma_{k+1}. \tag{11}$$

Здесь $\| M \|_2 = \max_{\|x\|=1} \| Mx \| = \sigma_1(M)$ — **спектральная норма** матрицы M (равна её наибольшему сингулярному числу). Утверждение остаётся в силе и для *фробениусовой* нормы $\| M \|_F = \sqrt{\sum_{i,j} m_{ij}^2}$:

$$\| A - A_k \|_F = \sqrt{\sigma_{k+1}^2 + \sigma_{k+2}^2 + \ldots + \sigma_r^2}. \tag{12}$$

Смысл теоремы: *если хочется сжать матрицу до ранга k, лучшее, что вы можете сделать, — оставить первые k компонент SVD*. Ошибка контролируется *отброшенным* сингулярным числом σ_{k+1}.

Пример: сжатие изображения

Возьмём фотографию 1024×768 в градациях серого. Хранение «в лоб» — это $1024 \cdot 768 = 786\,432$ чисел. Применим к матрице фото SVD и оставим только k старших компонент: придётся хранить

$$k \cdot (1024 + 768) + k = k \cdot 1793 \tag{13}$$

чисел — по одному столбцу u_i, одной строке $v_i^\top$ и одному числу σ_i на каждое слагаемое. Уже при $k = 50$ это $\sim 90\,000$ чисел — в *восемь* раз меньше исходного, а визуально потеря почти незаметна (рис. 0.3).

```
  (
 sequence(
   styled(child: [import], ..),
   styled(child: [ numpy ], ..),
   styled(child: [as], ..),
   styled(child: [ np], ..),
 ),
 sequence(
   styled(child: [import], ..),
   styled(child: [ matplotlib.pyplot ], ..),
   styled(child: [as], ..),
   styled(child: [ plt], ..),
 ),
 sequence(
   styled(child: [from], ..),
   styled(child: [ skimage ], ..),
   styled(child: [import], ..),
```

```
  styled(child: [ data, color], ..),
),
[],
styled(child: [# Load and convert to grayscale], ..),
sequence(
  styled(child: [img ], ..),
  styled(child: [=], ..),
  styled(child: [ color.rgb2gray(data.astronaut())        ], ..),
  styled(child: [# shape (512, 512)], ..),
),
sequence(
  styled(child: [U, S, Vt ], ..),
  styled(child: [=], ..),
  styled(child: [ np.linalg.svd(img, full_matrices], ..),
  styled(child: [=], ..),
  styled(child: [False], ..),
  styled(child: [)], ..),
),
[],
styled(child: [# Reconstruct using only top-k singular components], ..),
sequence(
  styled(child: [ranks ], ..),
  styled(child: [=], ..),
  styled(child: [ [], ..),
  styled(child: [1], ..),
  styled(child: [, ], ..),
  styled(child: [5], ..),
  styled(child: [, ], ..),
  styled(child: [20], ..),
  styled(child: [, ], ..),
  styled(child: [50], ..),
  styled(child: [, ], ..),
  styled(child: [200], ..),
  styled(child: []], ..),
),
sequence(
  styled(child: [fig, axes ], ..),
  styled(child: [=], ..),
  styled(child: [ plt.subplots(], ..),
  styled(child: [1], ..),
  styled(child: [, ], ..),
  styled(child: [len], ..),
  styled(child: [(ranks)], ..),
  styled(child: [+], ..),
  styled(child: [1], ..),
  styled(child: [, figsize], ..),
  styled(child: [=], ..),
  styled(child: [(], ..),
  styled(child: [14], ..),
  styled(child: [, ], ..),
  styled(child: [3], ..),
  styled(child: [))], ..),
),
sequence(
  styled(child: [axes[], ..),
  styled(child: [0], ..),
  styled(child: [].imshow(img, cmap], ..),
  styled(child: [=], ..),
```

```
    styled(child: ['gray'], ..),
    styled(child: [)], ..),
    styled(child: [;], ..),
    styled(child: [ axes[], ..),
    styled(child: [0], ..),
    styled(child: [].set_title(], ..),
    styled(child: ['original'], ..),
    styled(child: [)], ..),
  ),
  sequence(
    styled(child: [for], ..),
    styled(child: [ ax, k ], ..),
    styled(child: [in], ..),
    styled(child: [ ], ..),
    styled(child: [zip], ..),
    styled(child: [(axes[], ..),
    styled(child: [1], ..),
    styled(child: [:], ranks):], ..),
  ),
  sequence(
    styled(child: [    A_k ], ..),
    styled(child: [=], ..),
    styled(child: [ U[:, :k] ], ..),
    styled(child: [@], ..),
    styled(child: [ np.diag(S[:k]) ], ..),
    styled(child: [@], ..),
    styled(child: [ Vt[:k, :]], ..),
  ),
  sequence(
    styled(child: [    ax.imshow(A_k, cmap], ..),
    styled(child: [=], ..),
    styled(child: ['gray'], ..),
    styled(child: [)], ..),
  ),
  sequence(
    styled(child: [    ax.set_title(], ..),
    styled(child: [f'rank ], ..),
    styled(child: [{], ..),
    styled(child: [k], ..),
    styled(child: [}], ..),
    styled(child: ['], ..),
    styled(child: [)], ..),
  ),
  sequence(
    styled(child: [for], ..),
    styled(child: [ ax ], ..),
    styled(child: [in], ..),
    styled(child: [ axes: ax.axis(], ..),
    styled(child: ['off'], ..),
    styled(child: [)], ..),
  ),
  sequence(
    styled(child: [plt.tight_layout()], ..),
    styled(child: [;], ..),
    styled(child: [ plt.show()], ..),
  ),
)
```

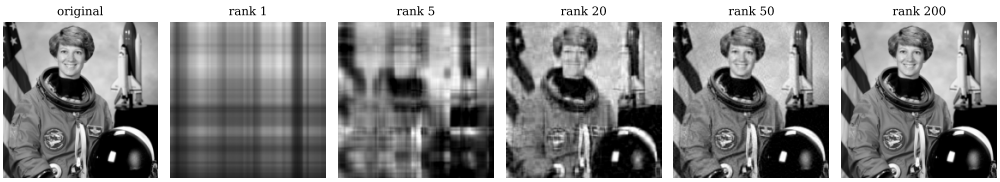

Рисунок 4. Рис. 0.3. Малоранговое приближение портрета астронавта. Уже при ранге 50 изображение визуально неотличимо от оригинала; при ранге 200 восстановление идеально. По теореме Эккарта–Янга это *лучшее* приближение фиксированного ранга.

А если картинка цветная? Цветное изображение $H \times W \times 3$ можно сжать тремя способами: (1) поканально, считая SVD отдельно для R, G, B; (2) перевести в *YCbCr* и сжимать только канал яркости Y; (3) «развернуть» изображение в матрицу $H \times 3W$ и применить одну SVD. В простейшем поканальном варианте для ранга r нужно хранить $3r(H + W + 1)$ чисел вместо $3HW$:

Рисунок 5. Рис. 0.4. Цветную фотографию сжимаем поканально: SVD ранга r для каждого из R, G, B и склейка. Под каждым кадром — доля памяти от оригинала. Варианты для цвета: t.me/fminxyz/16.

> **i** LoRA: малоранговая адаптация больших моделей
>
> В 2021 г. Эдвард Ху с соавторами заметили: когда большую языковую модель дообучают на новую задачу, *обновление* матриц весов ΔW оказывается почти малоранговой матрицей. Поэтому вместо обучения «полной» ΔW можно сразу искать её в виде $\Delta W = U\bar{V}$ с маленьким k, где U, V — тонкие прямоугольники.
>
> Это метод **LoRA** (*Low-Rank Adaptation*). Для модели с 175 миллиардами параметров такой приём сокращает объём дообучаемых весов в тысячи раз — и именно благодаря ему сегодня можно «дофайнтюнить» большую модель на бытовой видеокарте.
>
> Теоретический фундамент LoRA — та самая теорема Эккарта–Янга 0.3, которой почти 90 лет.

Eigenfaces: как алгоритм узнаёт лицо

Постановка задачи

В 1991 г. Мэттью Тёрк и Алекс Пентланд из MIT поставили простой вопрос: *можно ли обучить компьютер узнавать лицо, не прибегая ни к каким специальным «лицевым» признакам — носу, глазам, контурам, — а*

просто работая с матрицей пикселей? Их ответ — метод **собственных лиц** (*eigenfaces*) — стал классикой и открыл целую область computer vision.

Идея: каждая фотография лица — это точка в очень многомерном пространстве. Например, лицо 64×64 — это вектор длины $64 \cdot 64 = 4\,096$ в пространстве $\mathbb{R}^{4096}$. Двух разных людей снимали много раз — получаем облако точек в этом пространстве.

Главное наблюдение: *это облако очень узкое.* Хотя пространство $4\,096$ -мерное, реальные лица занимают в нём подпространство размерности всего лишь ~ 50. Найти это подпространство — и есть задача.

Рисунок 6. Рис. 0.5. Фрагмент датасета *Olivetti Faces*: 400 фотографий 64×64 пикселя, 40 человек по 10 снимков каждого. Каждая фотография — точка в $\mathbb{R}^{4096}$.

Шаг 1. Среднее лицо и центрирование

Пусть у нас N фотографий $x_1, \ldots, x_N \in \mathbb{R}^d$ (где $d = 4096$ для 64×64). Вычислим **среднее лицо**:

$$\bar{x} = \frac{1}{N} \sum_{i=1}^{N} x_i. \tag{14}$$

И вычтем его из каждого изображения: $\tilde{x}_i = x_i - \bar{x}$. *Среднее лицо* (рис. 0.6) — это размытая маска, в которой видно «лицо вообще»: усреднённые глаза, нос, рот, овал. Дальше мы будем работать только с отклонениями от этого среднего.

Рисунок 7. Рис. 0.6. Среднее лицо по 400 снимкам датасета Olivetti.

Шаг 2. Главные направления изменчивости

Соберём центрированные изображения в строки матрицы $X \in \mathbb{R}^{N \times d}$ (одна строка = одно лицо). Применим к ней SVD:

$$X = U \, \Sigma \, \bar{V}. \tag{15}$$

Каждая *строка* $\bar{v}_i$ матрицы $\bar{V}$ — это вектор длины $d = 4096$, который можно нарисовать как картинку 64×64. Эти картинки и называются **собственными лицами** — eigenfaces (рис. 0.7). Содержательно:

- v_1 — направление наибольшей изменчивости. Чаще всего это «освещение слева/справа».
- v_2 — направление второй по величине изменчивости (часто «улыбка / не улыбка»).
- $v_3, \ldots$ — постепенно более тонкие признаки: очки, форма щёк, наклон головы, индивидуальные особенности.

Рисунок 8. Рис. 0.7. Первые 12 eigenfaces датасета Olivetti, отрисованные как картинки 64×64. Эти странные «полу-лица» — координатные оси в пространстве, где живут все настоящие лица.

Шаг 3. Лицо как вектор из 50 чисел

Зафиксируем k (обычно $k = 30 \div 100$) и спроектируем каждое лицо x на эти k собственных лиц:

$$\alpha_i = \bar{v}_i (x - \bar{x}), \qquad i = 1, \ldots, k. \tag{16}$$

Получили вектор коэффициентов $\alpha = (\alpha_1, \dots, \alpha_k) \in \mathbb{R}^k$. Это и есть «лицо человека» с точки зрения алгоритма — *всего* 50 *чисел* вместо 4096.

Шаг 4. Распознавание и реконструкция

Распознавание. Чтобы узнать, кто на новой фотографии y, вычислим её коэффициенты β по (0.6) и найдём в базе ближайшего соседа: $\hat{\jmath} = \arg\min_j \| \beta - \alpha^{(j)} \|$.

Реконструкция. Зная α, можно восстановить лицо:

$$x \approx \bar{x} + \sum_{i=1}^{k} \alpha_i \, v_i. \tag{17}$$

Рис. 0.8 показывает, как качество реконструкции растёт с k.

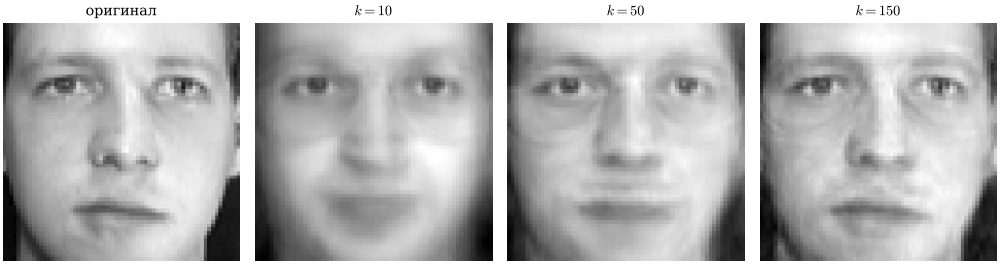

Рисунок 9. Рис. 0.8. Реконструкция одной из фотографий датасета через первые k eigenfaces. При $k = 10$ узнаётся «общий тип» лица; при $k = 50$ — уже конкретный человек; при $k = 150$ — почти оригинал.

Реализация на Python

```
(
 sequence(
  styled(child: [import], ..),
  styled(child: [ numpy ], ..),
  styled(child: [as], ..),
  styled(child: [ np], ..),
 ),
 sequence(
  styled(child: [import], ..),
  styled(child: [ matplotlib.pyplot ], ..),
  styled(child: [as], ..),
  styled(child: [ plt], ..),
 ),
 sequence(
  styled(child: [from], ..),
  styled(child: [ sklearn.datasets ], ..),
  styled(child: [import], ..),
  styled(child: [ fetch_olivetti_faces], ..),
 ),
 [],
 styled(child: [# 1. Load: 400 faces, 64x64 each], ..),
 sequence(
  styled(child: [faces ], ..),
  styled(child: [=], ..),
  styled(child: [ fetch_olivetti_faces().images       ], ..),
  styled(child: [# (400, 64, 64)], ..),
```

```
),
sequence(
  styled(child: [X ], ..),
  styled(child: [=], ..),
  styled(child: [ faces.reshape(], ..),
  styled(child: [400], ..),
  styled(child: [, ], ..),
  styled(child: [-], ..),
  styled(child: [1], ..),
  styled(child: []                  ], ..),
  styled(child: [# (400, 4096)], ..),
),
[],
styled(child: [# 2. Center: subtract the mean face], ..),
sequence(
  styled(child: [x_bar ], ..),
  styled(child: [=], ..),
  styled(child: [ X.mean(axis], ..),
  styled(child: [=], ..),
  styled(child: [0], ..),
  styled(child: [)], ..),
),
sequence(
  styled(child: [X_centered ], ..),
  styled(child: [=], ..),
  styled(child: [ X ], ..),
  styled(child: [-], ..),
  styled(child: [ x_bar], ..),
),
[],
styled(child: [# 3. SVD => eigenfaces are rows of Vt], ..),
sequence(
  styled(child: [U, S, Vt ], ..),
  styled(child: [=], ..),
  styled(child: [ np.linalg.svd(X_centered, full_matrices], ..),
  styled(child: [=], ..),
  styled(child: [False], ..),
  styled(child: [)], ..),
),
[],
styled(child: [# 4. Project all faces onto top-k eigenfaces], ..),
sequence(
  styled(child: [k ], ..),
  styled(child: [=], ..),
  styled(child: [ ], ..),
  styled(child: [50], ..),
),
sequence(
  styled(child: [alpha ], ..),
  styled(child: [=], ..),
  styled(child: [ X_centered ], ..),
  styled(child: [@], ..),
  styled(child: [ Vt[:k].T                ], ..),
  styled(child: [# (400, 50)], ..),
),
[],
styled(child: [# 5. Recognise: nearest neighbour in alpha-space], ..),
sequence(
```

```
  styled(child: [def], ..),
  styled(child: [ recognise(y, k], ..),
  styled(child: [=], ..),
  styled(child: [50], ..),
  styled(child: [):], ..),
 ),
 sequence(
  styled(child: [    y_centered ], ..),
  styled(child: [=], ..),
  styled(child: [ y.flatten() ], ..),
  styled(child: [-], ..),
  styled(child: [ x_bar], ..),
 ),
 sequence(
  styled(child: [    beta ], ..),
  styled(child: [=], ..),
  styled(child: [ Vt[:k] ], ..),
  styled(child: [@], ..),
  styled(child: [ y_centered], ..),
 ),
 sequence(
  styled(child: [    distances ], ..),
  styled(child: [=], ..),
  styled(child: [ np.linalg.norm(alpha ], ..),
  styled(child: [-], ..),
  styled(child: [ beta, axis], ..),
  styled(child: [=], ..),
  styled(child: [1], ..),
  styled(child: [)], ..),
 ),
 sequence(
  styled(child: [    ], ..),
  styled(child: [return], ..),
  styled(child: [ distances.argmin()], ..),
 ),
 [],
 styled(child: [# 6. Reconstruct a face from k coefficients], ..),
 sequence(
  styled(child: [def], ..),
  styled(child: [ reconstruct(idx, k], ..),
  styled(child: [=], ..),
  styled(child: [50], ..),
  styled(child: [):], ..),
 ),
 sequence(
  styled(child: [    ], ..),
  styled(child: [return], ..),
  styled(child: [ x_bar ], ..),
  styled(child: [+], ..),
  styled(child: [ Vt[:k].T ], ..),
  styled(child: [@], ..),
  styled(child: [ alpha[idx, :k]], ..),
 ),
)
```

Это интересно

Метод eigenfaces — родоначальник всех современных систем распознавания лиц, включая ту, что разблокирует ваш смартфон. Современные системы заменяют SVD на свёрточную нейросеть (см. § 3.3.7), но базовая идея — *сжать лицо в короткий вектор, в котором близкие лица оказываются рядом* — осталась той же. Эти короткие векторы сегодня называют **эмбеддингами**.

Eigencats: тот же приём для котов*

Метод eigenfaces работает на любых однородных объектах, не только на лицах. Возьмём *Cats Faces Dataset*: 29 843 фотографии котов 64×64 пикселя. Векторизуем — и получим матрицу $A \in \mathbb{R}^{29\,843 \times 4096}$. Сингулярные векторы матрицы A называют **eigencats**.

Рисунок 10. Рис. 0.9. Малоранговое восстановление четырёх котов на разных рангах r (анимация: t.me/fminxyz/12).

Как и в случае с лицами (§0.5), сами строки матрицы V образуют *базис* пространства котов. Каждая такая строка — собственный вектор размерности 4096, который можно нарисовать как картинку 64×64. Эти картинки и называют **eigencats**.

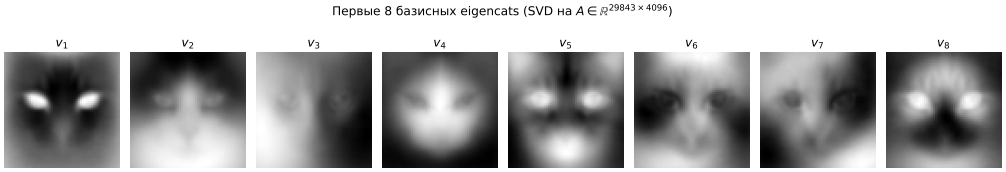

Рисунок 11. Рис. 0.10. Первые восемь eigencats — строки V, свёрнутые в квадраты 64×64. v_1 ловит освещение и общую форму морды, v_2 — контраст верх/низ (наклон), v_3 — лево/право, дальше — всё более тонкие особенности.

В сжатом виде каждый кот превращается в ~ 50 чисел: коэффициенты $\alpha_i = v_i(x - \bar{x})$. Этого хватает, чтобы потом восстановить узнаваемое изображение и отличать одного кота от другого.

i Что это нам говорит о ИИ

Пространство «реальных» объектов всегда узкое. Будь это лица людей, морды котов, рукописные цифры (датасет MNIST) или фотографии природы — реальные данные занимают в исходном многомерном пространстве *маленькое* подпространство. Распознавание = найти это подпространство и работать в нём.

Эта идея — фундамент современных автокодировщиков и эмбеддингов (см. §3.4.3–3.4.6). SVD — её самая ранняя и самая понятная реализация.

PageRank: как Google нашёл порядок в Интернете

Задача и история

К 1996 г. поисковики типа *AltaVišta* и *Yahoo!* ранжировали страницы по тому, сколько раз искомое слово встречается в тексте. Это легко обманывалось: достаточно было набить страницу популярными словами белым цветом по белому фону, и она попадала в топ выдачи.

Лоуренс Пейдж и Сергей Брин, тогда аспиранты Стэнфорда, предложили другой принцип ранжирования: *страница важна, если на неё ссылаются другие важные страницы.*

Граф ссылок как матрица

Обозначим N — число всех страниц Интернета (на 1998 г. — около $2,4 \cdot 10^8$, сегодня — $\sim 10^{11}$). Построим матрицу $P \in \mathbb{R}^{N\times N}$:

$$P_{ij}=\begin{cases}\frac{1}{c_j}, \text{если со страницы } \ j \ \text{ есть ссылка на } \ i \ , \\ 0, \ \text{ иначе,}\end{cases} \tag{18}$$

где c_j — *полное число исходящих ссылок* со страницы j. Содержательно: P_{ij} — это вероятность того, что пользователь, сидящий на странице j и случайно ткнувший на одну из её ссылок, окажется на странице i. Поэтому P называется матрицей переходов **случайного блуждания** по графу Интернета.

Стационарное распределение = собственный вектор

Пусть $r \in \mathbb{R}^N$ — вектор «важностей»: r_i — степень важности страницы i. Принцип Пейджа–Брина формализуется системой:

$$r = Pr. \tag{19}$$

То есть важность каждой страницы равна сумме важностей тех страниц, которые на неё ссылаются, делённой на число их исходящих ссылок.

Уравнение (0.7) говорит: r — это *собственный вектор* матрицы P с собственным значением 1. И в этом — вся суть PageRank.

⚠ Теорема 0.4. Перрон–Фробениус для PageRank

Если все элементы матрицы P положительны (или хотя бы граф ссылок сильно связен и непериодичен), то собственное значение 1 у P существует, оно *простое* и наибольшее по модулю, а соответствующий собственный вектор r единственен и имеет положительные компоненты. Этот r называется **стационарным распределением** случайного блуждания.

(Эта теорема — Оскар Перрон, 1907 г., и Георг Фробениус, 1912 г. — исторически появилась задолго до Интернета, в контексте моделей популяций.)

Демпфирующий множитель и формула Брина–Пейджа

В реальной матрице Интернета у некоторых страниц нет исходящих ссылок (*dangling pages*), а в других группах — циклы. Чтобы гарантировать применимость теоремы Перрона–Фробениуса, Пейдж и Брин ввели **демпфирующий множитель** $\beta \in (0, 1)$ (обычно $\beta = 0,85$):

$$r = \beta P r + \frac{1-\beta}{N}\,\mathbf{1}. \tag{20}$$

Здесь $\mathbf{1} = (1, 1, \ldots, 1)^\top$. Содержательно: с вероятностью $\beta \approx 0,85$ случайный сёрфер кликает по ссылке, а с вероятностью $1-\beta \approx 0,15$ — телепортируется на случайную страницу Интернета.

Как находить r: степенной метод

Прямо решать систему $(I - \beta P)r = \frac{1-\beta}{N}\mathbf{1}$ с $N \sim 10^{11}$ невозможно. Но и не надо: вектор r можно получить простой итерацией.

! Алгоритм

Степенной метод для PageRank.

1. Инициализация: $r^{(0)} \leftarrow \frac{1}{N}\mathbf{1}$.
2. Итерация: $r^{(t+1)} \leftarrow \beta\, P\, r^{(t)} + \frac{1-\beta}{N}\,\mathbf{1}$.
3. Остановка: $\| r^{(t+1)} - r^{(t)} \| < \varepsilon$.

Сходимость гарантируется теоремой Перрона–Фробениуса: $\| r^{(t)} - r \| \le C \cdot \beta^t$, то есть погрешность убывает геометрически со знаменателем $\beta = 0,85$. На практике хватает $50 \div 100$ итераций.

Игрушечный пример: 6 страниц

Возьмём маленький «Интернет» из шести страниц (рис. 0.11) и посчитаем для него PageRank.

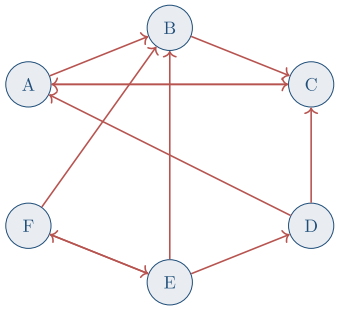

Рисунок 12. Рис. 0.11. Игрушечный «Интернет» из 6 страниц со ссылками. Видно, например, что на страницу C ссылаются три другие, а на F — только одна; будет ли C лидером по PageRank?

```
(
  sequence(
    styled(child: [import], ..),
    styled(child: [ numpy ], ..),
    styled(child: [as], ..),
    styled(child: [ np], ..),
  ),
  [],
  styled(child: [# Adjacency list -> column-stochastic transition matrix P], ..),
  sequence(
    styled(child: [links ], ..),
    styled(child: [=], ..),
    styled(child: [ {], ..),
    styled(child: ['A'], ..),
    styled(child: [: [], ..),
    styled(child: ['B'], ..),
    styled(child: [,], ..),
    styled(child: ['C'], ..),
    styled(child: [], ], ..),
```

```
    styled(child: ['B'], ..),
    styled(child: [: [], ..),
    styled(child: ['C'], ..),
    styled(child: [], ], ..),
    styled(child: ['C'], ..),
    styled(child: [: [], ..),
    styled(child: ['A'], ..),
    styled(child: [],], ..),
  ),
  sequence(
    styled(child: [          ], ..),
    styled(child: ['D'], ..),
    styled(child: [: [], ..),
    styled(child: ['C'], ..),
    styled(child: [,], ..),
    styled(child: ['A'], ..),
    styled(child: [], ], ..),
    styled(child: ['E'], ..),
    styled(child: [: [], ..),
    styled(child: ['D'], ..),
    styled(child: [,], ..),
    styled(child: ['B'], ..),
    styled(child: [,], ..),
    styled(child: ['F'], ..),
    styled(child: [], ], ..),
    styled(child: ['F'], ..),
    styled(child: [: [], ..),
    styled(child: ['E'], ..),
    styled(child: [,], ..),
    styled(child: ['B'], ..),
    styled(child: []}], ..),
  ),
  sequence(
    styled(child: [pages ], ..),
    styled(child: [=], ..),
    styled(child: [ ], ..),
    styled(child: [list], ..),
    styled(child: [(links)], ..),
    styled(child: [;], ..),
    styled(child: [ n ], ..),
    styled(child: [=], ..),
    styled(child: [ ], ..),
    styled(child: [len], ..),
    styled(child: [(pages)], ..),
    styled(child: [;], ..),
    styled(child: [ idx ], ..),
```

```
  styled(child: [=], ..),
  styled(child: [ {p: i ], ..),
  styled(child: [for], ..),
  styled(child: [ i, p ], ..),
  styled(child: [in], ..),
  styled(child: [ ], ..),
  styled(child: [enumerate], ..),
  styled(child: [(pages)}], ..),
),
[],
sequence(
  styled(child: [P ], ..),
  styled(child: [=], ..),
  styled(child: [ np.zeros((n, n))], ..),
),
sequence(
  styled(child: [for], ..),
  styled(child: [ j, p ], ..),
  styled(child: [in], ..),
  styled(child: [ ], ..),
  styled(child: [enumerate], ..),
  styled(child: [(pages):], ..),
),
sequence(
  styled(child: [    out ], ..),
  styled(child: [=], ..),
  styled(child: [ links[p]], ..),
),
sequence(
  styled(child: [    ], ..),
  styled(child: [for], ..),
  styled(child: [ q ], ..),
  styled(child: [in], ..),
  styled(child: [ out:], ..),
),
sequence(
  styled(child: [       P[idx[q], j] ], ..),
  styled(child: [=], ..),
  styled(child: [ ], ..),
  styled(child: [1.0], ..),
  styled(child: [ ], ..),
  styled(child: [/], ..),
  styled(child: [ ], ..),
  styled(child: [len], ..),
  styled(child: [(out)], ..),
),
```

```
[],
styled(child: [# PageRank via power iteration], ..),
sequence(
  styled(child: [beta, eps ], ..),
  styled(child: [=], ..),
  styled(child: [ ], ..),
  styled(child: [0.85], ..),
  styled(child: [, ], ..),
  styled(child: [1e-10], ..),
),
sequence(
  styled(child: [r ], ..),
  styled(child: [=], ..),
  styled(child: [ np.ones(n) ], ..),
  styled(child: [/], ..),
  styled(child: [ n], ..),
),
sequence(
  styled(child: [for], ..),
  styled(child: [ t ], ..),
  styled(child: [in], ..),
  styled(child: [ ], ..),
  styled(child: [range], ..),
  styled(child: [(], ..),
  styled(child: [200], ..),
  styled(child: [):], ..),
),
sequence(
  styled(child: [    r_new ], ..),
  styled(child: [=], ..),
  styled(child: [ beta ], ..),
  styled(child: [*], ..),
  styled(child: [ P ], ..),
  styled(child: [@], ..),
  styled(child: [ r ], ..),
  styled(child: [+], ..),
  styled(child: [ (], ..),
  styled(child: [1], ..),
  styled(child: [ ], ..),
  styled(child: [-], ..),
  styled(child: [ beta) ], ..),
  styled(child: [/], ..),
  styled(child: [ n], ..),
),
sequence(
  styled(child: [    ], ..),
```

```
  styled(child: [if], ..),
  styled(child: [ np.linalg.norm(r_new ], ..),
  styled(child: [-], ..),
  styled(child: [ r, ], ..),
  styled(child: [1], ..),
  styled(child: [) ], ..),
  styled(child: [<], ..),
  styled(child: [ eps: ], ..),
  styled(child: [break], ..),
),
sequence(
  styled(child: [    r ], ..),
  styled(child: [=], ..),
  styled(child: [ r_new], ..),
),
[],
sequence(
  styled(child: [for], ..),
  styled(child: [ p, ri ], ..),
  styled(child: [in], ..),
  styled(child: [ ], ..),
  styled(child: [sorted], ..),
  styled(child: [(], ..),
  styled(child: [zip], ..),
  styled(child: [(pages, r), key], ..),
  styled(child: [=], ..),
  styled(child: [lambda], ..),
  styled(child: [ t: ], ..),
  styled(child: [-], ..),
  styled(child: [t[], ..),
  styled(child: [1], ..),
  styled(child: []):], ..),
),
sequence(
  styled(child: [    ], ..),
  styled(child: [print], ..),
  styled(child: [(], ..),
  styled(child: [f'], ..),
  styled(child: [{], ..),
  styled(child: [p], ..),
  styled(child: [}], ..),
  styled(child: [: ], ..),
  styled(child: [{], ..),
  styled(child: [ri], ..),
  styled(child: [:.4f}], ..),
  styled(child: ['], ..),
```

```
        styled(child: [])], ..),
    ),
)
```

Выдача программы:

$$\text{C: } 0,295, \quad \text{A: } 0,252, \quad \text{B: } 0,175, \quad \text{E: } 0,098, \quad \text{F: } 0,092, \quad \text{D: } 0,087. \tag{0.27}$$

Лидер — C, хотя на неё формально ссылаются столько же страниц, сколько на A: PageRank учёл, что одна из ссылок на C приходит с уже-важной страницы B.

> **Историческая справка**
>
> Алгоритм PageRank был опубликован в 1998 г. в студенческой работе *Sergey Brin, Larry Page, «The PageRank Citation Ranking: Bringing Order to the Web»*. В том же году Брин и Пейдж основали Google. К 2004 г. компания вышла на IPO, а к 2010-м стала одной из крупнейших в мире. Сам Стэнфорд получил ∼ 336 миллионов долларов от лицензирования патента — который изначально просто описывал итерационный метод для собственного вектора.
>
> Идея использовать собственный вектор графа для измерения «важности» позднее была обобщена и применена в наукометрии (*Eigenfactor* для ранжирования научных журналов), в рекомендательных системах, а также в современных нейросетях для графов (*Graph Neural Networks*).

Когда нейросеть устойчива: оценки Липшицевости⋆

Сюжет: когда панда становится гиббоном

В 2014 г. Иэн Гудфеллоу и соавторы опубликовали ставшую знаменитой картинку (рис. 0.12): обычная фотография панды, которую современная нейросеть уверенно опознаёт как «panda» с вероятностью $57,7\%$. К каждому пикселю прибавлена крошечная, визуально незаметная человеку поправка. На обновлённой картинке сеть теперь уверенно (с вероятностью $99,3\%$) считает, что видит *гиббона*.

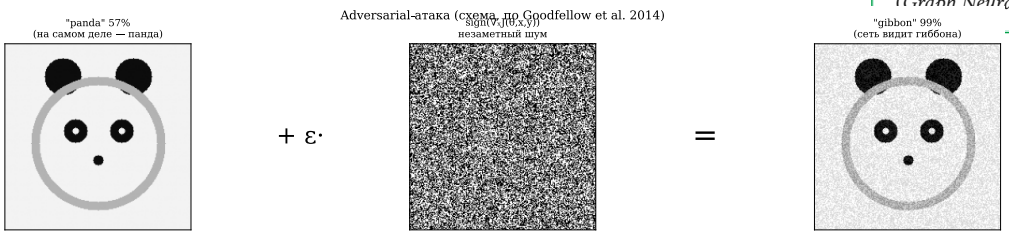

Рисунок 13. Рис. 0.12. Adversarial-атака на нейросеть (Goodfellow et al., 2014). Слева — оригинал, классифицирован как «panda» с уверенностью $57,7\%$. В центре — почти невидимое возмущение, масштабированное для наглядности. Справа — атакованный пример, классифицируется как «gibbon» с уверенностью $99,3\%$.

Этот эффект называется **adversarial-атакой** и долгое время считался экзотикой. Сейчас понятно: причина в простом числе, которое сопровождает любую функцию — её *константе Липшица*.

Константа Липшица: формальное определение

Определение 0.5. Константа Липшица функции

Функция $f : \mathbb{R}^n \to \mathbb{R}^m$ называется **Липшицевой** с константой $L > 0$, если для любых $x, y \in \mathbb{R}^n$ выполнено

$$\| f(x) - f(y) \| \le L \| x - y \| . \tag{22}$$

Наименьшее L, для которого (0.9) выполнено, называется *константой Липшица* функции f и обозначается $\mathrm{Lip}(f)$.

Содержательно $\mathrm{Lip}(f)$ показывает, во сколько раз может *усилиться* входное возмущение на выходе функции. Если $\mathrm{Lip}(f) = L$, то малое возмущение δ на входе может вызвать возмущение до $L \cdot \| \delta \|$ на выходе.

Пример. Для линейного отображения $f(x) = Wx$ имеем $f(x) - f(y) = W(x - y)$, откуда $\| f(x) - f(y) \| \le \| W \|_2 \cdot \| x - y \|$, и константа Липшица — это в точности спектральная норма матрицы W:

$$\mathrm{Lip}(Wx) = \| W \|_2 = \sigma_1(W). \tag{23}$$

Здесь $\sigma_1(W)$ — наибольшее сингулярное число матрицы W. Вот *здесь* в нашу историю входит SVD из § 0.3.

Композиция: Липшицева константа цепочки

Теорема 0.6. Композиция Липшицевых функций

Если f и g — Липшицевы с константами L_f и L_g, то их композиция $f \circ g$ — тоже Липшицева, причём

$$\mathrm{Lip}(f \circ g) \le \mathrm{Lip}(f) \cdot \mathrm{Lip}(g). \tag{24}$$

Доказательство. $\| f(g(x)) - f(g(y)) \| \le L_f \| g(x) - g(y) \| \le L_f L_g \| x - y \|$. ▫▫

Применение к нейросети

Возьмём типичную нейросеть глубины K:

$$f(x) = \sigma\big(W_K \, \sigma(W_{K-1} \, \sigma(\cdots \sigma(W_1 x)) \big), \tag{25}$$

где W_i — матрицы весов, σ — поэлементная активация (ReLU, сигмоида, и т.п.). Для большинства активаций $\mathrm{Lip}(\sigma) \le 1$ (для ReLU и сигмоиды это равно 1). По теореме 0.6:

$$\mathrm{Lip}(f) \le \prod_{i=1}^{K} \| W_i \|_2 \tag{26}$$

Это и есть тот «лишний рычаг», который объясняет уязвимость нейросетей. Если $\| W_i \|_2 \approx 2$ для каждого слоя из $K = 20$ слоёв, то возмущение на входе *потенциально* усиливается в $2^{20} \approx 10^6$ раз. Невидимая поправка δ с $\| \delta \| = 10^{-3}$ может вызвать изменение выхода до 10^3 — этого хватит, чтобы перебросить классификатор из одного класса в другой.

Как защищаться: спектральная нормализация

Естественная стратегия: после каждого шага обучения нормализовать матрицу W_i так, чтобы $\| W_i \|_2 = 1$. Это и есть метод **Spectral Normalization** (Миято и соавт., 2018), сделавший обучение генеративных нейросетей (GAN) намного стабильнее.

Технически: на каждом шаге считаем $\sigma_1(W)$ (с помощью одной итерации степенного метода — почти бесплатно!) и заменяем $W \mapsto W / \sigma_1(W)$.

```
  (
 sequence(
   styled(child: [def], ..),
   styled(child: [ spectral_normalize(W, n_iter], ..),
   styled(child: [=], ..),
   styled(child: [1], ..),
   styled(child: ]:], ..),
 ),
 sequence(
   styled(child: [    ], ..),
   styled(child: ["""One step of power iteration to estimate sigma_1(W) and rescale."""], ..),
 ),
 sequence(
   styled(child: [    u ], ..),
   styled(child: [=], ..),
   styled(child: [ np.random.randn(W.shape[], ..),
   styled(child: [0], ..),
   styled(child: [])], ..),
 ),
 sequence(
   styled(child: [    ], ..),
   styled(child: [for], ..),
   styled(child: [ _ ], ..),
   styled(child: [in], ..),
   styled(child: [ ], ..),
   styled(child: [range], ..),
   styled(child: [(n_iter):], ..),
 ),
 sequence(
   styled(child: [        v ], ..),
   styled(child: [=], ..),
   styled(child: [ W.T ], ..),
```

```
  $styled(child: [@], ..),
  $styled(child: [ u], ..),
  $styled(child: [;], ..),
  $styled(child: [ v ], ..),
  $styled(child: [/=], ..),
  $styled(child: [ np.linalg.norm(v)], ..),
 ),
 sequence(
  $styled(child: [        u ], ..),
  $styled(child: [=], ..),
  $styled(child: [ W ], ..),
  $styled(child: [@], ..),
  $styled(child: [ v], ..),
  $styled(child: [;], ..),
  $styled(child: [    u ], ..),
  $styled(child: [/=], ..),
  $styled(child: [ np.linalg.norm(u)], ..),
 ),
 sequence(
  $styled(child: [     sigma_1 ], ..),
  $styled(child: [=], ..),
  $styled(child: [ u ], ..),
  $styled(child: [@], ..),
  $styled(child: [ W ], ..),
  $styled(child: [@], ..),
  $styled(child: [ v], ..),
 ),
 sequence(
  $styled(child: [    ], ..),
  $styled(child: [return], ..),
  $styled(child: [ W ], ..),
  $styled(child: [/], ..),
  $styled(child: [ sigma_1], ..),
 ),
)
```

> **Это интересно**
>
> **Связь с устойчивостью в физике.** Похожая ситуация хорошо известна в теории динамических систем: если константа Липшица отображения > 1, малое возмущение начального условия экспоненциально нарастает (классический эффект бабочки в погодных моделях). Spectral Normalization можно рассматривать как искусственное гашение этого эффекта.

QR-алгоритм: как находят все *собственные числа**

Зачем нужен ещё один метод

В §0.7 мы использовали *степенной метод*, чтобы найти *один* собственный вектор — доминантный, соответствующий наибольшему по модулю собственному числу. А что, если нужны *все* собственные числа сразу — например, чтобы найти главные компоненты k-ранговой аппроксимации, посчитать спектральную норму или определить устойчивость линейной системы $\dot{x} = Ax$ (она устойчива, когда все собственные числа имеют отрицательную действительную часть)?

Здесь и нужен **QR-алгоритм** — одна из жемчужин численных методов. Описан в 1961 году Джоном Фрэнсисом и независимо Верой Кубланской, по сей день остаётся *рабочей лошадкой* библиотек LAPACK и NumPy — именно его вы вызываете каждый раз, когда пишете numpy.linalg.eig.

Сам алгоритм: две строчки

Идея до неприличия проста. Возьмём QR-разложение матрицы A: $A = QR$, где Q ортогональная (поворот), R верхнетреугольная. Теперь переставим сомножители:

$$A_{k+1} = R_k Q_k, \quad \text{где} \quad Q_k R_k = \mathrm{qr}(A_k). \tag{27}$$

И повторим. Всё.

! QR-алгоритм

Вход: квадратная матрица $A_0 \in \mathbb{R}^{n\times n}$, число итераций N.

Повторять $k = 0, 1, \ldots, N-1$:

1. Вычислить QR-разложение: $Q_k, R_k \leftarrow \mathrm{qr}(A_k)$.
2. Обновить: $A_{k+1} \leftarrow R_k Q_k$.

Возврат: A_N — (почти) верхнетреугольная; на её диагонали стоят собственные числа исходной A.

Что происходит и почему это работает

Все матрицы A_k *подобны* исходной: $A_{k+1} = \bar{Q}_k A_k Q_k$. А подобные матрицы имеют одинаковые собственные числа (это базовый факт линейной алгебры). При этом с каждой итерацией поддиагональные элементы A_k уменьшаются в геометрической прогрессии, и в пределе

A_k становится верхнетреугольной — значит, на её диагонали как раз и стоят собственные числа.

⚠ Теорема 0.7. Сходимость QR-алгоритма

Пусть собственные числа A действительны и различны по модулю: $|\lambda_1| > |\lambda_2| > \ldots > |\lambda_n|$. Тогда последовательность A_k, полученная QR-итерациями, сходится к верхнетреугольной матрице, на диагонали которой стоят $\lambda_1, \lambda_2, \ldots, \lambda_n$ (в порядке убывания модуля), причём поддиагональные элементы убывают как $|\lambda_{i+1}/\lambda_i|^k$.

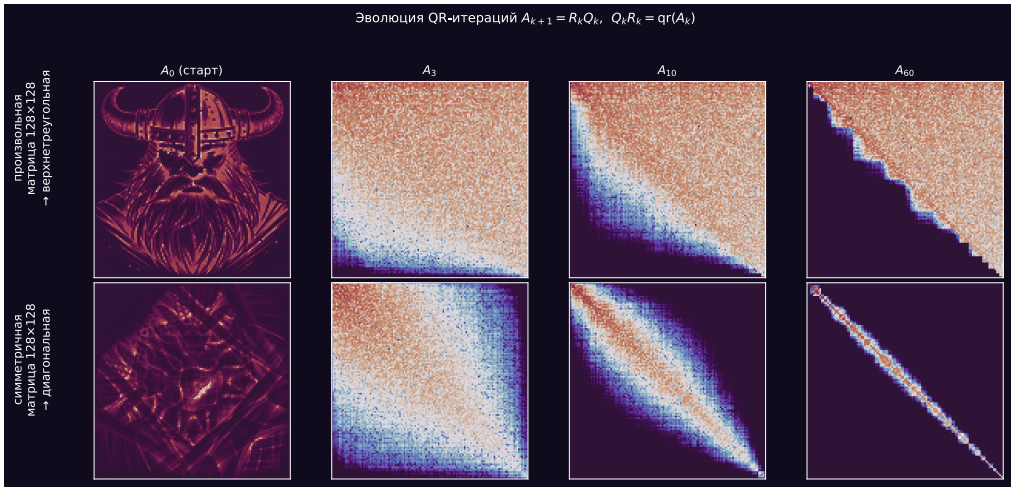

Рисунок 14. Рис. 0.13. Сходимость QR-алгоритма на 5×5-матрицах. Верхний ряд: произвольная (несимметричная) матрица сходится к верхнетреугольной с собственными числами на диагонали $(4, -3, 2.5, -1.5, 1)$. Нижний ряд: симметричная матрица сходится к *диагональной* — это полезное свойство симметрии. Источник: t.me/fminxyz/30.

Симметричный случай: сходимся к диагональной

Если исходная A симметрична, $\overline{A} = A$, то все промежуточные A_k тоже симметричны, и сходимость идёт не просто к верхнетреугольной, а сразу к *диагональной* матрице. На рис. 0.13 (нижний ряд) видно, что внедиагональные элементы зануляются с обеих сторон одновременно. Это особенно ценно: получив диагональ, мы сразу имеем *и* собственные числа, *и* собственные векторы (произведение всех Q_k сходится к матрице собственных векторов).

```
  (
 sequence(
   styled(child: [import], ..),
   styled(child: [ numpy ], ..),
   styled(child: [as], ..),
   styled(child: [ np], ..),
 ),
 [],
 sequence(
```

```
  styled(child: [def], ..),
  styled(child: [ qr_eigvals(A, n_iters], ..),
  styled(child: [=], ..),
  styled(child: [200], ..),
  styled(child: [):], ..),
),
sequence(
  styled(child: [    A ], ..),
  styled(child: [=], ..),
  styled(child: [ A.astype(], ..),
  styled(child: [float], ..),
  styled(child: [).copy()], ..),
),
sequence(
  styled(child: [    ], ..),
  styled(child: [for], ..),
  styled(child: [ _ ], ..),
  styled(child: [in], ..),
  styled(child: [ ], ..),
  styled(child: [range], ..),
  styled(child: [(n_iters):], ..),
),
sequence(
  styled(child: [        Q, R ], ..),
  styled(child: [=], ..),
  styled(child: [ np.linalg.qr(A)], ..),
),
sequence(
  styled(child: [        A ], ..),
  styled(child: [=], ..),
  styled(child: [ R ], ..),
  styled(child: [@], ..),
  styled(child: [ Q], ..),
),
sequence(
  styled(child: [    ], ..),
  styled(child: [return], ..),
  styled(child: [ np.diag(A)            ], ..),
  styled(child: [# собственные числа], ..),
),
[],
styled(child: [# Проверка на случайной симметричной матрице:], ..),
sequence(
  styled(child: [np.random.seed(], ..),
  styled(child: [0], ..),
  styled(child: [)], ..),
),
sequence(
  styled(child: [M ], ..),
  styled(child: [=], ..),
  styled(child: [ np.random.randn(], ..),
  styled(child: [5], ..),
  styled(child: [, ], ..),
  styled(child: [5], ..),
  styled(child: [)], ..),
),
sequence(
  styled(child: [S ], ..),
```

```
    styled(child: [=], ..),
    styled(child: [ (M ], ..),
    styled(child: [+], ..),
    styled(child: [ M.T) ], ..),
    styled(child: [/], ..),
    styled(child: [ ], ..),
    styled(child: [2], ..),
  ),
  sequence(
    styled(child: [print], ..),
    styled(child: [(], ..),
    styled(child: [sorted], ..),
    styled(child: [(qr_eigvals(S),              reverse], ..),
    styled(child: [=], ..),
    styled(child: [True], ..),
    styled(child: [))], ..),
  ),
  sequence(
    styled(child: [print], ..),
    styled(child: [(], ..),
    styled(child: [sorted], ..),
    styled(child: [(np.linalg.eigvalsh(S).tolist(), reverse], ..),
    styled(child: [=], ..),
    styled(child: [True], ..),
    styled(child: [))], ..),
  ),
)
```

> **i** Реальный QR в LAPACK
>
> В практических реализациях используют *двухступенчатый* вариант: сначала A приводят к **форме Хессенберга** (почти треугольной — только одна поддиагональ ненулевая) ортогональными преобразованиями Хаусхолдера, а потом запускают QR-итерации со сдвигами (*Wilkinson shift*). Это даёт сходимость за $\mathcal{O}(n)$ итераций вместо $\mathcal{O}(n \log n)$ <<наивного>> QR, и общая сложность получается $\mathcal{O}(n^3)$ — тот же порядок, что и у SVD.

Связь с SVD

QR и SVD — родственники. Сингулярные числа матрицы A совпадают с квадратными корнями собственных чисел матрицы $\bar{A}A$. Поэтому один из стандартных способов вычислить SVD — сначала построить $\bar{A}A$ (она симметрична), применить QR-алгоритм и получить $\sigma_i = \sqrt{\lambda_i}$. На практике, однако, так *не делают*: построение $\bar{A}A$ возводит в квадрат число обусловленности и теряет точность. Современные алгоритмы SVD (например, *Голуба–Кахана*) применяют QR-итерации напрямую к би-диагональной форме A, обходя $\bar{A}A$.

Подведение итогов главы

- Матрица — это *универсальный носитель данных* для ИИ: картинки, тексты, графы, веса нейросети.
- **Числа с плавающей точкой** (§0.2) — вычисления на компьютере всегда приближённые с относительной ошибкой $\sim \varepsilon_{\text{маш}} \approx 10^{-16}$. Самая опасная ошибка — *катастрофическое сокращение* при вычитании близких чисел; её часто можно устранить, переписав формулу алгебраически эквивалентным способом.
- **Сингулярное разложение** $A = U\Sigma\bar{V}$ (§0.3) представляет любую матрицу как «поворот → растяжение по осям → поворот». Сингулярные числа σ_i — это коэффициенты растяжения.
- **Теорема Эккарта–Янга** (§0.4): $A_k = \sum_{i \le k} \sigma_i u_i^\top v_i$ — лучшее приближение ранга k матрицы A. На этой теореме стоит сжатие изображений, метод eigenfaces и современный LoRA для дообучения больших языковых моделей.
- **Eigenfaces / eigencats** (§0.5–0.6): реальные данные (лица, котики, цифры) живут в очень узком подпространстве своего номинального многомерного пространства. Найти это подпространство = применить SVD к выборке.
- **PageRank** (§0.7): «важность» страницы Интернета = компонента собственного вектора матрицы переходов случайного блуждания. Алгоритм находит его простой итерацией; теорема Перрона–Фробениуса гарантирует сходимость.
- **Константа Липшица** нейросети (§0.8) растёт мультипликативно от слоя к слою; именно поэтому глубокие сети уязвимы к adversarial-атакам. Спектральная нормализация ограничивает $\|W_i\|_2 = 1$ и стабилизирует обучение.

Что почитать дальше

- *Е. Е. Тыртышников.* Методы численного анализа. М.: Академия, 2007. — классический российский учебник по вычислительной линейной алгебре университетского уровня.
- *G. Strang.* Linear Algebra and Learning from Data. Wellesley-Cambridge Press, 2019. — свежая книга от автора знаменитого MIT-курса 18.06, написанная с прицелом на data science.

- *L. N. Trefethen, D. Bau.* Numerical Linear Algebra. SIAM, 1997. — классика, с очень понятным изложением SVD и итерационных методов.
- *Видеокурс «Numerical Linear Algebra»*, И. В. Оселедец, Сколтех (github.com/MerkulovDaniil/nla360). Лекции, семинары, ноутбуки — из которых, в частности, и взяты eigencats для этой главы.
- *S. Brin, L. Page.* The Anatomy of a Large-Scale Hypertextual Web Search Engine. WWW Conference, 1998. — оригинальная статья про PageRank, читается легко.
- *I. Goodfellow et al.* Explaining and Harnessing Adversarial Examples. ICLR, 2015. — статья про панду-гиббона, из которой взят рис. 0.12.

Большой итоговый проект

Постройте свою рекомендательную систему для фильмов.

Возьмите датасет MovieLens-100K (открытый, 100 000 оценок, ~ 1000 пользователей $\times \sim 1700$ фильмов). Постройте матрицу «пользователь–фильм» R, где $R_{ij} \in \{1, \dots, 5\}$ — оценка, поставленная пользователем i фильму j, а 0 — «не смотрел».

1. Постройте малоранговое приближение $R \approx U \Sigma \bar{V}$ с $k = 20$ компонентами через SVD.
2. Объясните содержательно, что такое строки U (профили пользователей) и столбцы V (профили фильмов) в этом приближении. Сколько чисел теперь хранит модель вместо $1000 \cdot 1700$?
3. Для нескольких реальных пользователей выпишите топ-5 фильмов, которые рекомендация модели предлагает посмотреть. Совпадают ли они с любимыми жанрами пользователя?
4. Найдите два «противоположных» фильма с наибольшим расстоянием между их столбцами V и подумайте, можно ли увидеть содержательный смысл в этой разделяющей оси (боевик vs. мелодрама? старое vs. новое?).
5. ($\star$) Замените SVD на алгоритм PageRank, построив граф «фильмы, которые часто оценивают одинаково». Получится ли тот же топ?

! Задачи для самостоятельной работы

1. Сложите в Python $10\,000\,000$ копий числа $0,1$. Сколько должно получиться? Сколько получается на самом деле? Почему?
2. Вычислите $f(x) = \sqrt{x+1} - \sqrt{x}$ при $x = 10^{16}$ напрямую и через формулу $f(x) = \frac{1}{\sqrt{x+1}+\sqrt{x}}$. Сравните ответы. Какой устойчив, почему?
3. Найдите вручную SVD матрицы $A = \begin{pmatrix} 3 & 0 \\ 0 & 4 \end{pmatrix}$ и $A = \begin{pmatrix} 0 & 1 \\ -1 & 0 \end{pmatrix}$. Что в обоих случаях геометрически делают U, Σ, V?
4. Возьмите свою фотографию 300×300 в градациях серого. Сделайте сжатие через SVD с рангами $k = 5, 20, 50, 100$. При каком k вы перестаёте узнавать самого себя?
5. Для игрушечного графа из рис. 0.11 прокрутите 5 итераций степенного метода *вручную*, начав с $r^{(0)} = (\frac{1}{6}, \dots, \frac{1}{6})$. Совпали ли ваши значения с программными после 5 шагов? а после 20?
6. ⋆ Покажите, что для функции $\sigma(x) = \max(0, x)$ (ReLU) константа Липшица равна 1. Тот же вопрос для сигмоиды $\sigma(x) = 1/(1+e^{-x})$: чему равна $\mathrm{Lip}(\sigma)$ и в какой точке производная максимальна?
7. ⋆ Найдите спектральную норму матрицы $W = \begin{pmatrix} 1 & 2 \\ 2 & 1 \end{pmatrix}$ двумя способами: (а) через определение $\| W \|_2 = \max_{\|x\|=1} \| Wx \|$ (с использованием собственных значений $W^{\top}W$); (б) одной итерацией степенного метода (с любым стартом). Совпали ли ответы?