Материалы кафедры ММП факультета ВМК МГУ. Введение в глубокое обучение.

Занятие 03. Автоматическое дифференцирование

Занятие провел: Оганов Александр (@welmud)

Материалы составил: Оганов Александр (@welmud)

Материалы основаны на лекции Кропотова Дмитрия Александровича

Москва, Весенний семестр 2026

Источники:

- Лекция Кропотова Дмитрия Александровича на курсе "Практикум на ЭВМ" кафедры ММП
- Дополнительный конспект прошлых лет

На этом занятии мы познакомимся с основными методами дифференцирования на компьютере, обсудим какие методы применяются на практике и почему. Рассмотрим отдельные примеры матрично-векторного дифференцирования: градиенты через решение системы линейных уравнений, как оптимизировать сквозь методы оптимизации и как можно использовать сингулярные числа для регуляризации.

Автоматическое дифференцирование

Основная тема занятия: "Как на компьютере вычислять производные функции?"

Мы предполагаем, что функция реализована на питоне и мы умеем ее вычислять, а целью является нахождение градиента, то есть производной функции по всем параметрам. Нахождение градиента необходимая задача при обучении нейронных сетей. На практике, при построении нейронной сети, мы реализуем только проход вперед (`forward pass`) – то как вход преобразуется через последовательность слоев, а подсчет градиента реализуется автоматически.

С точки зрения методов оптимизации существуют подходы безградиентной оптимизации (методы нулевого порядка), но обычно безградиентная оптимизация требует на порядки больше шагов для сходимости. Если ваша функция гладкая и есть возможность подсчета градиента, то крайне желательно этим воспользоваться.

Помимо `автоматического дифференцирования` есть другой подход к подсчету производной на компьютере с помощью разностного дифференцирования.

Пусть есть функция $f(x)$ (можно вычислить в любой точке), тогда производную по направлению d можно приблизить следующим образом:

$$\nabla f(x)^T d = \frac{f(x + \varepsilon d) - f(x)}{\varepsilon},$$

где ε близкое к нулю число. Следовательно, если есть процедура вычисления функции f, то можно высчитать производную для каждого параметра нейронной сети.

Вопрос: Почему этот способ не используется на практике для вычисления градиента при обучении нейронной сети?

Ответ:

Благодаря разностной схемы мы находим производную по направлению, то есть для подсчёта градиента по N параметрам нужно сделать $2N$ вычислений функции, что на практике невозможно. Соврменные нейронные сети имеют более миллиарда параметров, для этого есть более продвинутые техники.

Данный способ хорошо подходит для проверки корректности подсчета градиента, но не пригоден для использования во время обучения.

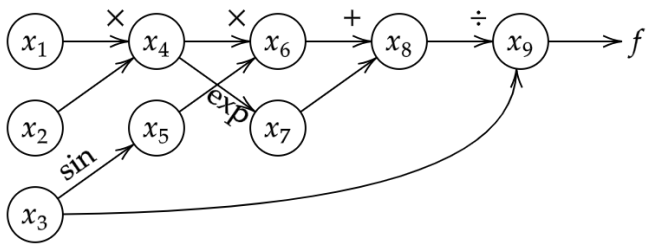
Вместо разностного подхода используют методы `автоматического дифференцирования` , то есть вычисление производной по графу вычислений.

Пример графа вычислений

Рассмотрим следующую функцию:

$$f(x_1, x_2, x_3) = \frac{x_1 x_2 \sin(x_3) + exp(x_1 x_2)}{x_3}.$$

Первый шаг: по формуле построить граф вычислений, пример графа вычислений, который реализует предложенную функцию f представлен ниже



Вопрос: Однозначно ли задается граф вычислений по формуле, которой записана функция?

Ответ:

Так как наша главная задача вычислить производную на компьютере, то есть мы описываем вычисление функции на Python, а значит по нашему коду и строиться граф вычислений. Задачи оптизации и построения графа вычислений похожи на задачи компиляторов, которые по коду строят внутреннее представление программы. Оптимизация графа вычислений может значительно ускорить код.

Вычисление производных по графу

Наша основная цель вычислить градиент функции, то есть посчитать производные f по независимым переменным x_1, x_2, x_3 с помощью автоматического дифференцирования.

В общем случае выделяют 3 режима работы:

- вычисления производной проходом вперед (`forward mode`)
- вычисления производной проходом назад (`backward mode`)
- смешанный

Проход вперед для вычисления производной

Следуя названию метода мы будем вычислять производную слева направо, соответственно мы будем пересчитывать:

$$\frac{\partial x_i}{\partial x_j}, \quad i = 1, \dots, 9, \quad j = 1, 2, 3.$$

Соответственно проинициализируем значения на первом слое:

$$\frac{\partial x_1}{\partial x_1} = 1, \frac{\partial x_2}{\partial x_1} = 0, \frac{\partial x_3}{\partial x_1} = 0.$$

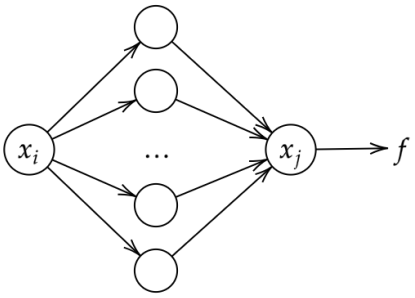
Дальше мы делаем шаг вперед, например, вычисляем x_4 и сразу считаем производные, используя производные с прошлого слоя.

$$\begin{aligned} x_4 &= x_1 x_2, \\ \frac{\partial x_4}{\partial x_1} &= \frac{\partial x_1}{\partial x_1} x_2 + x_1 \frac{\partial x_2}{\partial x_1}. \end{aligned}$$

Для x_5 мы получим следующее:

$$\begin{aligned} x_5 &= \sin(x_3), \\ \frac{\partial x_5}{\partial x_1} &= \cos(x_3) \frac{\partial x_3}{\partial x_1}. \end{aligned}$$

В общем случае для графа:



Мы получим:

$$\frac{\partial x_j}{\partial x_i} = \sum_{k:(k,j)\in\Sigma} \frac{\partial x_j}{\partial x_k} \frac{\partial x_k}{\partial x_i},$$

где запись $(k, j) \in \Sigma$ означает, что есть ребро из k в j. $\frac{\partial x_k}{\partial x_i}$ мы знаем с прошлой итерации, а $\frac{\partial x_j}{\partial x_k}$ сможем вычислить на ходу. Заметим, что нам требуется хранить производные только с прошлого слоя, а при переходе на новый слой мы можем забыть старый.

Вопрос: Какие недостатки вы видите при вычислении производной в режиме прохода вперед (`forward mode`)?

Ответ:

В области обычно считают, что вычисление производной имеет порядок сложности сравнимый с вычислением функции (с точки зрения формул). Например, посчитать $\sin$ также трудно, как и посчитать $\cos$. Текущий метод для вычисления производной по входу имеет сложность схожую со сложностью подсчёта функции. При входе размера

N потребуется N раз вычислить функцию. При этом, данный метод крайне эффективен если вход имеет маленькую размерность, а выход большую, например, векторозначные функции со скалярным аргументом.

Проход назад для вычисления производной

Как следует из названия мы будем вычислять производную справа налево, соответственно нас будут интересовать:

$$\frac{\partial f}{\partial x_j}, \quad j = 1, \dots, 9.$$

Соответственно проинициализируем значения:

$$\frac{\partial f}{\partial x_9} = 1.$$

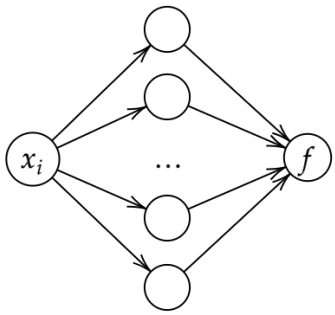
Далее идем налево:

$$\frac{\partial f}{\partial x_8} = \frac{\partial f}{\partial x_9}\frac{\partial x_9}{\partial x_8} = \frac{\partial f}{\partial x_9}\frac{1}{x_3}.$$

Рассмотрим произвольный шаг, например, найдем $\frac{\partial f}{\partial x_4}$, следовательно, мы уже нашли $\frac{\partial f}{\partial x_6}$ и $\frac{\partial f}{\partial x_7}$, воспользуемся этим:

$$\frac{\partial f}{\partial x_4} = \frac{\partial f}{\partial x_6}\frac{\partial x_6}{\partial x_4} + \frac{\partial f}{\partial x_7}\frac{\partial x_7}{\partial x_4}.$$

В общем случае для графа:



Мы получим:

$$\frac{\partial f}{\partial x_j} = \sum_{k:(j,k)\in\Sigma} \frac{\partial f}{\partial x_k}\frac{\partial x_k}{\partial x_j}.$$

Заметим, что после работы алгоритма мы получаем полный вектор градиента, следовательно, мы рассчитываем все необходимое за один проход.

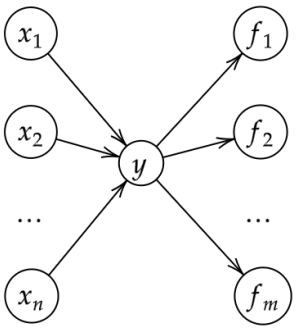
Вопрос: Крайне редко алгоритм, который смотрит на задачу с другой стороны может работать эффективней без какой-либо платы. За счет чего была достигнута эффективность?

Ответ:

Для прохода назад нам необходимо сохранить в памяти выходы всех слоев (активаций), чтобы эффективно считать производные считать выхода слоя по входу. Мы получили эффективность по времени, но появились дополнительные расходы по памяти. Существующие библиотеки, например, PyTorch автоматически сохраняют в графе вычислений, поэтому часто выделяется дополнительная память при вычислении функции потерь. Если у вас нет необходимости в подсчёте градиентов, например, на валидации, то лучше явно отключить сохранение активаций. Например, поэтому на валидации можно использовать батч большего размера, чем на обучении.

Смешанный режим вычисления производной

Рассмотрим следующий граф, где y скалярная переменная:



Нас интересует матрица якобиана, то есть $\frac{\partial f_j}{\partial x_i}$.

Вопрос: в каком режиме получится максимально эффективно вычислить якобиан?

Мы можем воспользоваться структурой графа и вычислить левую часть графа проходом назад, а правую проходом вперед. Мы получим, что $\frac{\partial f_j}{\partial x_i} = \frac{\partial f_j}{\partial y}\frac{\partial y}{\partial x_i}$, то есть суммарно прошли по графу вычислений один раз. Такой подход основан на анализе графа вычислений, есть отдельные конференции посвященные вычислению и поиску специальных подграфов для их эффективного вычисления. В целом задача прохода по графу вычислений очень схожа на работу компилятора и применяемых в нем оптимизаций. Задача перевода кода и поиск шаблонов для эффективного перевода в машинный код схожа на задачу `автоматического дифференцирования`.

На практике мы крайне редко столкнемся с использованием прямого и смешанного режима, так как в основных фреймворках все написано для прохода назад. Например, PyTorch только недавно научился поддерживать forward mode.

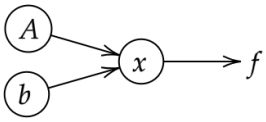
Матрично-векторное дифференцирование

Выше мы разобрали примеры, в которых участвовали скалярные функции, при этом на практике мы в основном работаем с матрицами. Теперь мы разберем более интересные примеры.

Градиенты через решение СЛАУ

Допустим у нас есть СЛАУ $Ax = b$, где $A \in \mathbb{R}^{n \times n}$, $\det(A) \neq 0$, дальше для подсчета лосса модели мы как-то используем решение x. Цель найти градиент функции потерь по матрице A и по вектору b, то есть найти $\nabla_A f$ и $\nabla_b f$.

Для начала составим граф вычислений:



Мы будем пользоваться свойством инвариантности формы первого дифференциала, то есть верно:

$$\mathrm{d}f = \nabla_x f^T \mathrm{d}x,$$

если мы рассматриваем функцию f в параметризации x, и верно:

$$\mathrm{d}f = \nabla_b f^T \mathrm{d}b + \big(\nabla_A f, \mathrm{d}A\big),$$

если мы смотрим на f в параметризации A, b (так как x зависит от A, b).

Для нахождения градиентов мы распишем $\mathrm{d}x$ через $\mathrm{d}A$ и $\mathrm{d}b$ сгруппируем слагаемые, соответсвующее коэффициенты и будут необходимыми градиентами.

Вопрос: Как найти $\mathrm{d}x$?

Ответ:

У нас есть только одно уравнение, дающее связь между A, b и x, продифференцируем его:

$$
\begin{aligned}
&Ax = b, \\
&\mathrm{d}Ax + A\mathrm{d}x = \mathrm{d}b, \\
&\mathrm{d}x = A^{-1}(\mathrm{d}b - \mathrm{d}Ax).
\end{aligned}
$$

Подставим dx и получим:

$$
\begin{aligned}
&\mathrm{d}f = \nabla_x f^T \mathrm{d}x = \nabla_x f^T A^{-1}(\mathrm{d}b - \mathrm{d}Ax) = \\
&= \nabla_x f^T A^{-1}\mathrm{d}b - \nabla_x f^T A^{-1}\mathrm{d}Ax = \nabla_b f^T \mathrm{d}b + \left(\nabla_A f, \mathrm{d}A\right).
\end{aligned}
$$

Откуда находим:

$$
\nabla_b f = A^{-T}\nabla_x f.
$$

Вспомним, что $\mathrm{d}f$ число, а значит $\nabla_x f^T A^{-1}\mathrm{d}Ax$ тоже должно быть числом, следовательно:

$$
\nabla_x f^T A^{-1}\mathrm{d}Ax = tr(\nabla_x f^T A^{-1}\mathrm{d}Ax) = tr(x\nabla_x f^T A^{-1}\mathrm{d}A) = \left(A^{-T}\nabla_x f x^T, \mathrm{d}A\right).
$$

Откуда получим:

$$
\nabla_A f = -A^{-T}\nabla_x f x^T = -\nabla_b f x^T.
$$

Для нахождения градиента по A, надо знать градиент по b, который является решением СЛАУ $A^T\nabla_b f = \nabla_x f$. Мы получили, что вычисление производной схоже порядком сложности на вычисление функции (в обоих случаях надо решать СЛАУ).

Так как мы не привязывали алгоритм к конкретному методу получения x из A и b, мы можем обсудить как на практике можно решать СЛАУ.

Вопрос: Какие методы решения СЛАУ вы знаете?

Ответ:

Воспользуемся LU-разложением, тогда получим:

$$
\begin{aligned}
&A = LU, \\
&Ax = b \Rightarrow LUx = b, \\
&Ly = b, Ux = y, \\
&y = L^{-1}b, x = U^{-1}y, \\
&x = U^{-1}L^{-1}b.
\end{aligned}
$$

Обратить треугольную матрицу можно, например, методом Гаусса за $O(n^2)$, то есть для подсчета производной мы можем считать эффективно, не зная, как получалось x (возможно и за $O(n^3)$).

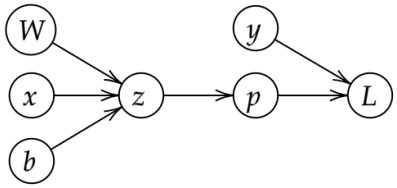
Логистическая регрессия

Рассмотрим логистическую регрессию, пусть $x \in \mathbb{R}^n$, $y \in \{1, 2, \dots, k\}$, $z = Wx + b$, где $W \in \mathbb{R}^{k \times d}$. Для вычисления функции потерь нам нужно знать вероятности каждого класса, поэтому обозначим их за p, где $p = \frac{\exp(z)}{\exp(z)^T 1}$ ($exp(z)$ поэлементная функция, 1 вектор из всех единиц).

Вопрос: Как выглядит функция потерь для многоклассовой классификации?

Ответ:

Функцию потерь можно записать как $L(p, y) = -\log p^T 1_y$. Теперь когда мы все определили, запишем граф вычислений:



Для $\mathrm{d}L$ выполнено следующее:

$$\mathrm{d}L = \nabla_p L^T \mathrm{d}p = -(\frac{1}{p^T 1_y})1_y \mathrm{d}p.$$

Пользуясь инвариантностью первого дифференциала, мы знаем, что:

$$\mathrm{d}L = \nabla_p L^T \mathrm{d}p = \nabla_z L^T \mathrm{d}z.$$

То есть наша задача найти чему равно $\mathrm{d}p$, для этого продифференцируем p по определению:

$$\mathrm{d}p = \frac{diag(\exp(z))\mathrm{d}z \exp(z)^T 1 - \exp(z) 1^T diag(\exp(z))\mathrm{d}z}{(\exp(z)^T 1)^2} =$$
$$= \frac{1}{\exp(z)^T 1} diag(\exp(z))\mathrm{d}z - \frac{1}{(\exp(z)^T 1)^2} \exp(z) 1^T diag(\exp(z))\mathrm{d}z.$$

Подставим $\mathrm{d}p$ в выражение $\mathrm{d}L$ для нахождение $\nabla_z L$:

$$\mathrm{d}L = \nabla_z L^T \mathrm{d}z = \Big[\nabla_p L^T \frac{1}{\exp(z)^T 1} diag(\exp(z)) - \nabla_p L^T \frac{1}{(\exp(z)^T 1)^2} \exp(z) 1^T diag(\exp(z))\Big] \mathrm{d}z.$$

Тогда получим:

$$\nabla_z L = \frac{1}{\exp(z)^T 1} diag(\exp(z)) \nabla_p L - \frac{\nabla_p L^T \exp(z)}{(\exp(z)^T 1)^2} diag(\exp(z)) 1.$$

Для нахождения градиентов по параметрам z, заметим:

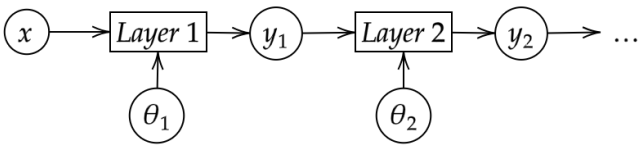
$$
\begin{aligned}
&z = Wx + b, \\
&\mathrm{d}z = \mathrm{d}Wx + W\mathrm{d}x + \mathrm{d}b, \\
&\nabla_z L^T dz = \nabla_z L^T \mathrm{d}Wx + \nabla_z L^T W \mathrm{d}x + \nabla_z L^T \mathrm{d}b.
\end{aligned}
$$

Откуда не трудно найти градиенты.

$$
\begin{aligned}
&\nabla_b L = \nabla_z L, \\
&\nabla_x L = W^T \nabla_z L, \\
&\nabla_W L = \nabla_z L x^T.
\end{aligned}
$$

Общий вид

Выше мы рассмотрели конкретные примеры, теперь изучим нейросеть как некую последовательность слоев в общем виде. Любую нейросеть можно представить в виде следующего графа.



Для автоматического дифференцирования необходимо реализовать две функции: `forward` и `backward`.

Функция `forward` задается как $y_{i+1} = forward(y_i, \theta_{i+1})$

Функция `backward` необходима для прохода назад. $backward(\nabla_{y_{i+1}} L, y_i, y_{i+1}, \theta_{i+1})$ вычисляет $\nabla_{y_i} L, \nabla_{\theta_{i+1}} L$

В большинстве случаев `PyTorch` сам реализует функцию `backward` по уже заданной `forward`. Однако бывают примеры, когда приходится самому явно реализовывать `backward`, например, при работе с комплексными эмбеддингами. Часто комплексные числа возникают при обработке сигналов, так как многие алгоритмы используют преобразование Фурье.

Другим примером служит более эффективная реализация `backward`. Иногда, вы знаете что определенные слои идут последовательно и разумней их объединить (использовать `fuse`) для более эффективного вычисления. Иногда слои важно объединить в один блок для вывода полезной теории, например, инициализация весов линейного слоя при учете функции активации. Более сложным примером является квантизация моделей. Подробнее с этими примерами мы познакомимся позже.

Градиенты через итерационный процесс

Первый наш пример был о том, как прогнать градиент через решение СЛАУ. Мы получили формулы, которые **никак не зависят** от способа получения решения. Мы бы могли залезть внутрь алгоритма, например, рассмотреть конкретный способ нахождения решения СЛАУ. Тогда можно было бы итерации численного метода рассмотреть как слои нейронной сети, как куски графа вычислений и гнать производную через граф вычислений стандартными методами. Такой подход требует дополнительной памяти (надо хранить промежуточные значения).

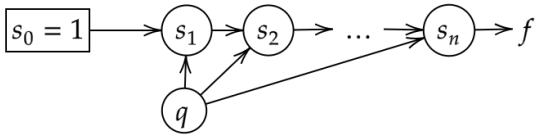
Мы рассмотрим частный случай, когда можно посмотреть на алгоритм, с которым работаем и воспользоваться его структурой, мы по сути будем инвертировать его итерации.

Рассмотрим пример: сумма членов геометрической прогрессии

Пусть $q \in (0, 1)$, $s_n = 1 + q + \ldots + q^n$ и s_n как-то используется для вычисления функции потерь $f(s_n)$. Заметим, что $s_n = 1 + q * s_{n-1}$.

Простой способ: записать аналитическую формулу и прогнать через нее градиенты. В этом случае мы не учитываем ошибки округления и численную стабильность, тем самым получаем смещенное значения градиента, что делает процедуру не корректной.

Второй способ: прогнать градиенты через итерации алгоритма. Для этого запишем граф вычислений.



Пусть мы знаем $\frac{\partial f}{\partial s_n}$, тогда мы можем записать:

$$\begin{aligned} \mathrm{d}f = \frac{\partial f}{\partial s_n}\mathrm{d}s_n = \\ = \frac{\partial f}{\partial s_n}(\mathrm{d}q s_{n-1} + q\mathrm{d}s_{n-1}). \end{aligned}$$

Откуда получаем:

$$\begin{aligned} \frac{\partial f}{\partial s_{n-1}} = \frac{\partial f}{\partial s_n}q, \\ \frac{\partial f}{\partial q} = \frac{\partial f}{\partial s_n}s_{n-1}. \end{aligned}$$

То есть формально, чтобы узнать $\frac{\partial f}{\partial q}$ необходимо вычислить s_{n-1} и так рекурсивно пройти по всему графу. С другой стороны мы знаем, что $s_{n-1} = \frac{s_n - 1}{q}$ (обратная итерация) и тогда:

$$\frac{\partial f}{\partial q} = \frac{\partial f}{\partial s_n} \frac{s_n - 1}{q}.$$

Такой способ не всегда работает, так как у нас была явная формула и мы смогли легко инвертировать. Например, если слой нейронной сети это метод оптимизации, то для `SGD` не существует метода инвертации. Примерами, когда мы можем инвертировать, являются `SGD` с моментумом и `Adam`. Подход с инвертацией позволяет значительно ускорить обучение.

Градиент через сингуряные числа

Рассмотрим матрицу $W \in \mathbb{R}^{n \times m}$ и ее сингулярное разножение (SVD) $W = U\Sigma V^T$, где $U^T U = I$ и $VV^T = I$, матрица Σ является диагональной, где на диагонали стоят сингулярные числа $\{\sigma_1, \dots, \sigma_{\min(n,m)}\}$

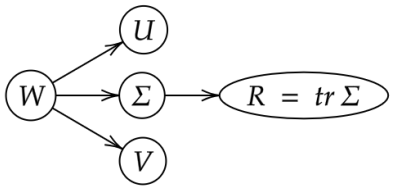
Вопрос: Зачем сингулярные числа могут быть полезны в нейронных сетях?

Часто, когда говорят про SVD, рассматривают ядерную норму матрицы, которую можно записать как $\|W\|_* = trace(\Sigma) = \sum \sigma_i$

Если к нашему лоссу мы добавим ядерную норму, то получится регуляризация, которая поощряет матрицы наименьшего ранга.

$$Loss(W, y, X) + \lambda \|W\|_* \to \min_W$$

Построим граф вычислений.



Наша цель найти $\nabla_W R$ (мы хотим обновить матрицу), а значит по аналогии с первым примером нам потребуется найти $\mathrm{d}\Sigma$.

$$\begin{aligned}
&W = U\Sigma V^T, \\
&\mathrm{d}W = \mathrm{d}U\Sigma V^T + U\mathrm{d}\Sigma V^T + U\Sigma \mathrm{d}V^T, \\
&U^T \mathrm{d}W V = U^T \mathrm{d}U\Sigma + \mathrm{d}\Sigma + \Sigma \mathrm{d}V^T V.
\end{aligned}$$

Рассмотрим $U^T \mathrm{d}U$, есть ощущение, что это должна быть какая-то особенная матрица, так как у нас есть ограничения на вид U.

$$U^T U = I,$$
$$\mathrm{d}U^T U + U^T \mathrm{d}U = 0.$$

Если мы обозначим за $C = \mathrm{d}U^T U$, то выше мы получили $C + C^T = 0$, следовательно, C кососимметричная матрица. Этот факт означает, что на диагонали у матрицы C стоят нули. Откуда получаем, что $(C\Sigma)_i i = 0$, так как Σ диагональная матрица. Кроме того мы знаем, что $\mathrm{d}\Sigma$ тоже диагональная, а значит:

$$\mathrm{d}\Sigma = diag(U^T \mathrm{d}W V - U^T \mathrm{d}U \Sigma - \Sigma \mathrm{d}V^T V) = diag(U^T \mathrm{d}W V).$$

Так как
$\mathrm{d}R = \left(\nabla_\Sigma R, \mathrm{d}\Sigma\right) = tr(\nabla_\Sigma R^T \mathrm{d}\Sigma) = tr(\nabla_\Sigma R^T diag(U^T \mathrm{d}W V)) = tr(\nabla_\Sigma R^T U^T \mathrm{d}W V$
, мы получим $\nabla_W R = U \nabla_\Sigma R V^T$.

Тем самым мы можем регуляризировать сверточные слои (поощрять матрицы с меньшим рангом). Ядерную норму используют на практике, например, в NVidia.