

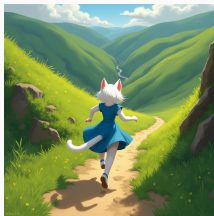
СТОХАСТИЧЕСКИЙ ГРАДИЕНТНЫЙ СПУСК

Сергей Николенко

СПбГУ — Санкт-Петербург

14 сентября 2024 г.

Random facts:



- 14 сентября 1812 г., через неделю после Бородинского сражения, в Москву прибыл Наполеон (и тут же начались пожары), а 14 сентября 1817 г. в Москву прибыл памятник Минину и Пожарскому
- 14 сентября 1917 г. Временное правительство объявило Россию республикой, упразднив каторгу и ссылку и объявив политическую амнистию
- 14 сентября 1927 г. в Ницце на колесо автомобиля намотался шарф Айседоры Дункан
- 14 сентября 1929 г. футболисты киевского «Динамо» провели свой первый международный матч, со сборной рабочих Нижней Австрии
- 14 сентября 1947 г. в Петродворце был снова открыт фонтан «Самсон»
- 14 сентября 1999 г. организаторы конкурса «Мисс Америка» разрешили участвовать в нём разведённым женщинам и тем, кто делал аборт
- 14 сентября 2010 г. Сенат Франции одобрил законопроект, запрещающий женщинам носить паранджу, чадру и никаб в публичных местах

ГРАДИЕНТНЫЙ СПУСК И ГРАФЫ ВЫЧИСЛЕНИЙ

- Градиентный спуск: считаем градиент относительно весов, двигаемся в нужном направлении.
- Формально: для функции ошибки E , целевых значений y и модели f с параметрами θ :

$$E(\theta) = \sum_{(\mathbf{x}, y) \in D} E(f(\mathbf{x}, \theta), y),$$

$$\theta_t = \theta_{t-1} - \eta \nabla E(\theta_{t-1}) = \theta_{t-1} - \eta \sum_{(\mathbf{x}, y) \in D} \nabla E(f(\mathbf{x}, \theta_{t-1}), y).$$

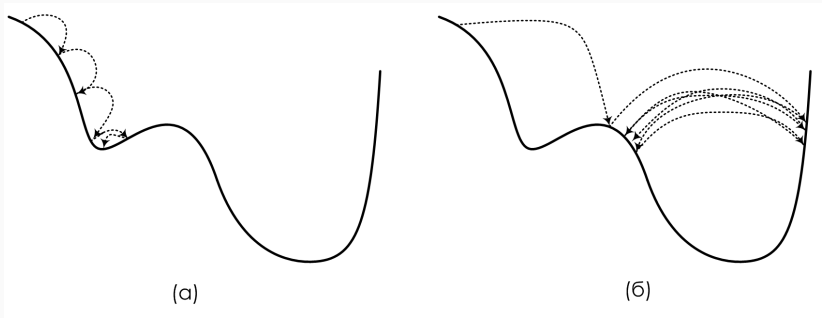
- То есть надо по всему датасету пройти, чтобы хоть куда-то сдвинуться?..

- Нет, конечно — *стохастический градиентный спуск* обновляет после каждого примера:

$$\theta_t = \theta_{t-1} - \eta \nabla E(f(\mathbf{x}_t, \theta_{t-1}), y_t),$$

- А на практике обычно используют *мини-батчи*, их легко параллелизовать и они сглаживают излишнюю “стохастичность”.
- Пока что единственный реальный параметр – это скорость обучения η .

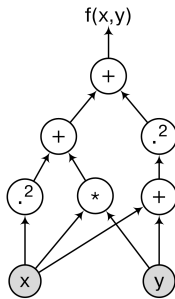
- Со скоростью обучения η масса проблем:



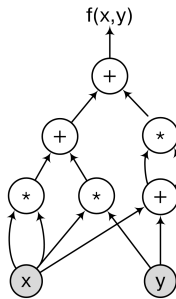
- Мы вернёмся к ним позже, а пока поговорим о производных.

ГРАФ ВЫЧИСЛЕНИЙ, FROP AND VPROP

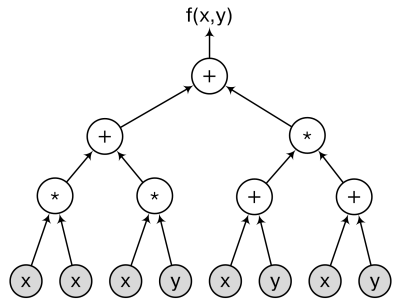
- Представим функцию как композицию простых функций (т.е. таких, от которых можно производную взять).
- Пример: $f(x, y) = x^2 + xy + (x + y)^2$:



(a)



(б)



(в)

- Градиент теперь можно взять по правилу дифференцирования сложной функции (chain rule):

$$(f \circ g)'(x) = (f(g(x)))' = f'(g(x))g'(x).$$

- По сути это значит, что небольшое изменение δx приводит к

$$\delta f = f'(g(x))\delta g = f'(g(x))g'(x)\delta x.$$

- Нам нужно только уметь брать *градиенты*, т.е. производные по векторам:

$$\nabla_{\mathbf{x}} f = \begin{pmatrix} \frac{\partial f}{\partial x_1} \\ \vdots \\ \frac{\partial f}{\partial x_n} \end{pmatrix}.$$

$$\nabla_{\mathbf{x}}(f \circ g) = \begin{pmatrix} \frac{\partial f \circ g}{\partial x_1} \\ \vdots \\ \frac{\partial f \circ g}{\partial x_n} \end{pmatrix} = \begin{pmatrix} \frac{\partial f}{\partial g} \frac{\partial g}{\partial x_1} \\ \vdots \\ \frac{\partial f}{\partial g} \frac{\partial g}{\partial x_n} \end{pmatrix} = \frac{\partial f}{\partial g} \nabla_{\mathbf{x}} g.$$

- А если f зависит от x несколько раз,
 $f = f(g_1(x), g_2(x), \dots, g_k(x))$, δx тоже несколько раз
появляется:

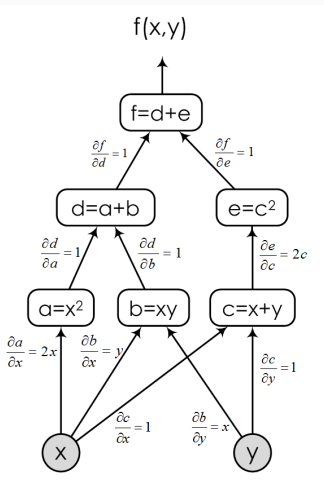
$$\frac{\partial f}{\partial x} = \frac{\partial f}{\partial g_1} \frac{\partial g_1}{\partial x} + \dots + \frac{\partial f}{\partial g_k} \frac{\partial g_k}{\partial x} = \sum_{i=1}^k \frac{\partial f}{\partial g_i} \frac{\partial g_i}{\partial x}.$$

$$\nabla_{\mathbf{x}} f = \frac{\partial f}{\partial g_1} \nabla_{\mathbf{x}} g_1 + \dots + \frac{\partial f}{\partial g_k} \nabla_{\mathbf{x}} g_k = \sum_{i=1}^k \frac{\partial f}{\partial g_i} \nabla_{\mathbf{x}} g_i.$$

- Это матричное умножение на матрицу Якоби:

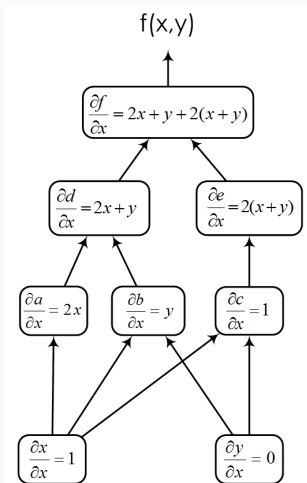
$$\nabla_{\mathbf{x}} f = \nabla_{\mathbf{x}} \mathbf{g} \nabla_{\mathbf{g}} f, \text{ где } \nabla_{\mathbf{x}} \mathbf{g} = \begin{pmatrix} \frac{\partial g_1}{\partial x_1} & \dots & \frac{\partial g_k}{\partial x_1} \\ \vdots & & \vdots \\ \frac{\partial g_1}{\partial x_n} & \dots & \frac{\partial g_k}{\partial x_n} \end{pmatrix}.$$

- Возвращаемся к примеру:

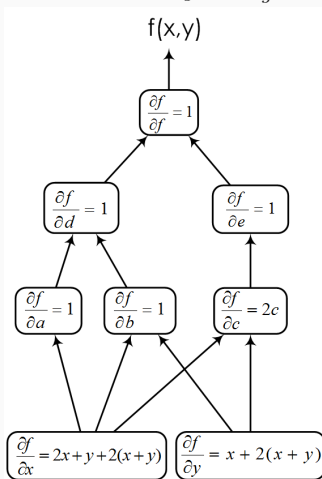


ГРАФ ВЫЧИСЛЕНИЙ, FROP AND BPROP

- Прямое распространение (forward propagation, fprop):
вычисляем $\frac{\partial f}{\partial x}$ как сложную функцию.

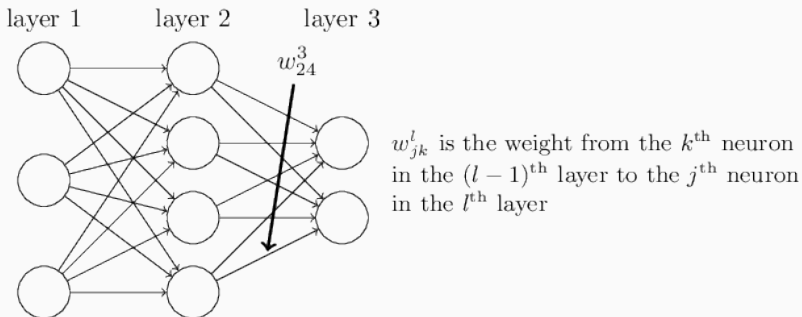


- Обратное распространение (backpropagation, backprop):
начинаем от конца и идём как $\frac{\partial f}{\partial g} = \sum_{g' \in \text{Children}(g)} \frac{\partial f}{\partial g'} \frac{\partial g'}{\partial g}$.



- Backprop гораздо лучше: получаем все производные за один проход по графу.
- Вот и всё! Теперь мы можем считать градиенты от любых, сколь угодно сложных функций; нужно только, чтобы они представлялись как композиции простых.
- А это всё, что нужно для градиентного спуска!!
- Библиотеки *pyTorch*, *TensorFlow*, *theano* — это на самом деле библиотеки для автоматического дифференцирования, это их основная функция.
- И теперь мы можем реализовать массу “классических” моделей в *pyTorch* и обучить их градиентным спуском.
- А у живых нейронов, кстати, не получается, потому что нужно два разных “алгоритма” для вычисления самой функции и градиента.

- Если сеть организована в слои, то fprop и backprop можно векторизовать, т.е. представить в виде матричных операций.



- Обозначим $w_{jk}^{(l)}$ вес связи от k -го нейрона слоя $(l-1)$ к j -му нейрону слоя l .

- Обозначим $w_{jk}^{(l)}$ вес связи от k -го нейрона слоя $(l - 1)$ к j -му нейрону слоя l .
- Также $b_j^{(l)}$ – bias j -го нейрона, $a_j^{(l)}$ – его активация, т.е.

$$a_j^{(l)} = h \left(\sum_k w_{jk}^{(l)} a_k^{(l-1)} + b_j^{(l)} \right).$$

- Это можно записать в векторной форме:

$$\mathbf{a}^{(l)} = h \left((\mathbf{w}^{(l)})^\top \mathbf{a}^{(l-1)} + \mathbf{b}^{(l)} \right) = h \left(\mathbf{z}^{(l)} \right).$$

- А наша цель – посчитать производные функции ошибки по всем переменным: $\frac{\partial C}{\partial w_{jk}^{(l)}}, \frac{\partial C}{\partial b_j^{(l)}}$.

- Определяем ошибку j -го нейрона в слое l :

$$\delta_j^{(l)} = \frac{\partial C}{\partial z_j^{(l)}}.$$

- И backprop теперь можно делать от последнего уровня L ко входу, сразу в векторном виде:

$$\delta_j^{(L)} = \frac{\partial C}{\partial a^{(L)}} h' \left(z_j^{(L)} \right), \text{ т.е. } \delta^{(L)} = \nabla_{\mathbf{a}^{(L)}} C \odot h' \left(\mathbf{z}^{(L)} \right),$$

$$\delta^{(l)} = \left((W^{(l+1)})^\top \delta^{(l+1)} \right) \odot h' \left(\mathbf{z}^{(l)} \right),$$

$$\frac{\partial C}{\partial b_j^{(l)}} = \delta_j^{(l)},$$

$$\frac{\partial C}{\partial w_{jk}^{(l)}} = a_k^{(l-1)} \delta_j^{(l)}.$$

- Это всё операции, которые легко параллелизовать на GPU.

- Backpropagation – идея со сложной судьбой.
- Конечно, это просто расчёт градиентов для градиентного спуска.
- Так что уже в 1960-х было понятно, как тащить производные динамическим программированием (и тем более удивительно насчёт Minsky, Papert).
- В явном виде полностью BP для нейронных сетей – M.Sc. thesis (Linnainmaa, 1970).
- (Hinton, 1974) – переоткрыл backprop, популяризовал.
- Rumelhart, Hinton, Williams, “Learning representations by back-propagating errors” (Nature, 1986).

ВАРИАНТЫ ГРАДИЕНТНОГО СПУСКА

- «Ванильный» градиентный спуск:

$$\mathbf{x}_k = \mathbf{x}_{k-1} - \alpha \nabla f(\mathbf{x}_k).$$

- Всё зависит от скорости обучения α .
- Первая мысль — пусть α уменьшается со временем:
 - линейно (linear decay):

$$\alpha = \alpha_0 \left(1 - \frac{t}{T}\right);$$

- или экспоненциально (exponential decay):

$$\alpha = \alpha_0 e^{-\frac{t}{T}}.$$

- Об этом есть большая наука. Например, условия Вольфе (Wolfe conditions): если мы решаем задачу минимизации $\min_{\mathbf{x}} f(\mathbf{x})$, и на шаге k уже нашли направление $\mathbf{p}_k$, в котором двигаться (например, $\mathbf{p}_k = \nabla_{\mathbf{x}} f(\mathbf{x}_k)$), т.е. надо решить $\min_{\alpha} f(\mathbf{x}_k + \alpha \mathbf{p}_k)$, то:
 - для $\phi_k(\alpha) = f(\mathbf{x}_k + \alpha \mathbf{p}_k)$ будет $\phi'_k(\alpha) = \nabla f(\mathbf{x}_k + \alpha \mathbf{p}_k)^{\top} \mathbf{p}_k$, и если $\mathbf{p}_k$ – направление спуска, то $\phi'_k(0) < 0$;
 - шаг α должен удовлетворять условиям Армихо (Armijo rule):

$$\phi_k(\alpha) \leq \phi_k(0) + c_1 \alpha \phi'_k(0) \text{ для некоторого } c_1 \in (0, \frac{1}{2});$$

- или даже более сильным условиям Вульфа (Wolfe rule): Армихо плюс

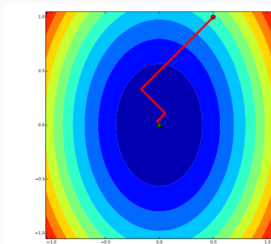
$$|\phi'_k(\alpha)| \leq c_2 |\phi'_k(0)|,$$

т.е. мы хотим уменьшить проекцию градиента.

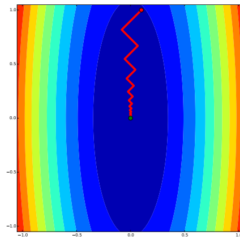
- Останавливаем когда $\|\nabla_{\mathbf{x}} f(\mathbf{x}_k)\|^2 \leq \epsilon$ или $\|\nabla_{\mathbf{x}} f(\mathbf{x}_k)\|^2 \leq \epsilon \|\nabla_{\mathbf{x}} f(\mathbf{x}_0)\|^2$ (а почему квадрат, кстати?).

ГРАДИЕНТНЫЙ СПУСК

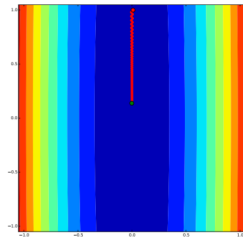
- Давайте посмотрим, что происходит, если масштаб разный:
для функции $f(x, y) = \frac{1}{2}x^2 + \frac{\rho}{2}y^2 \rightarrow \min_{x,y}$



$\rho = 0.5$



$\rho = 0.1$



$\rho = 0.01$

- Для вытянутых «долин» (переменных с разным масштабированием) мы сразу получаем кучу лишних итераций, очень медленно.
- Лучше быть *адаптивным*; как это сделать?

- Лучше всего, конечно, *метод Ньютона*: давайте отмасштабируем обратно при помощи гессиана

$$\mathbf{g}_k = \nabla_{\mathbf{x}} f(\mathbf{x}_k), \quad H_k = \nabla_{\mathbf{x}}^2 f(\mathbf{x}_k), \quad \text{и} \quad \mathbf{x}_{k+1} = \mathbf{x}_k - \alpha_k H_k^{-1} \mathbf{g}_k.$$

- Здесь тоже применимо условие Армихо:

$$\alpha_k : \quad f(\mathbf{x}_{k+1}) \leq f(\mathbf{x}_k) - c_1 \alpha_k \mathbf{g}_k^\top H_k^{-1} \mathbf{g}_k, \quad c_1 \approx 10^{-4}.$$

- Было бы круто! Но H_k посчитать просто нереально.

- Есть, правда, приближения.
- Метод сопряжённых градиентов, квази-ньютоновские методы...
- L-BFGS (limited memory Broyden–Fletcher–Goldfarb–Shanno):
 - строим аппроксимацию к H^{-1} ;
 - для этого сохраняем последовательно апдейты аргументов функции и градиентов и выражаем через них H^{-1} .
- Интересный открытый вопрос: можно ли заставить L-BFGS работать для deep learning?
- Но пока не получается, в основном потому, что всё-таки нужно уметь считать градиент.
- А ведь у нас обычно нет возможности даже градиент вычислить...

СТОХАСТИЧЕСКИЙ ГРАДИЕНТНЫЙ СПУСК

- У нас обычно стохастический градиентный спуск:

$$\mathbf{x}_t = \mathbf{x}_{t-1} - \alpha \nabla f(\mathbf{x}_t, \mathbf{x}_{t-1}, y_t).$$

- Да ещё и с мини-батчами; как это понять формально?
- Мы обычно решаем задачу *стохастической оптимизации*:

$$F(\mathbf{x}) = \mathbb{E}_{q(\mathbf{y})} f(\mathbf{x}, \mathbf{y}) \rightarrow \min_{\mathbf{x}} :$$

- минимизация эмпирического риска

$$F(\mathbf{x}) = \frac{1}{N} \sum_{i=1}^N f_i(\mathbf{x}) = \mathbb{E}_{i \sim U(1, \dots, N)} f_i(\mathbf{x}) \rightarrow \min_{\mathbf{x}};$$

- минимизация вариационной нижней оценки (ELBO)... но об этом позже.
- Что такое теперь, получается, мини-батчи?

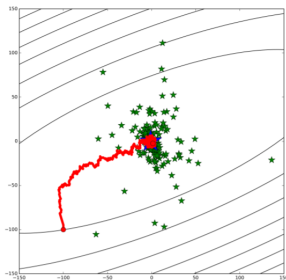
- Да просто эмпирические оценки общей функции по подвыборке:

$$\hat{F}(\mathbf{x}) = \frac{1}{m} \sum_{i=1}^m f(\mathbf{x}, \mathbf{y}_i), \quad \hat{\mathbf{g}}(\mathbf{x}) = \frac{1}{m} \sum_{i=1}^m \nabla_{\mathbf{x}} f(\mathbf{x}, \mathbf{y}_i).$$

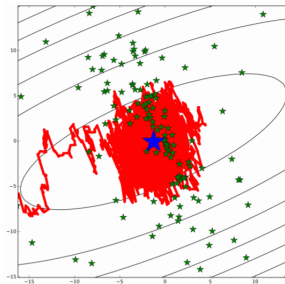
- Это очень хорошие оценки: несмещённые, сходятся на бесконечности (правда, медленно), легко посчитать.
- В целом, так и мотивируется стохастический градиентный спуск (SGD): метод Монте-Карло по сути.
- Но есть проблемы...

СТОХАСТИЧЕСКИЙ ГРАДИЕНТНЫЙ СПУСК

- Проблемы SGD:
 - никогда не идёт в правильном направлении,
 - шаг не равен нулю в оптимуме $F(\mathbf{x})$, т.е. не может сойтись с постоянной длиной шага,
 - мы не знаем ни $F(\mathbf{x})$, ни $\nabla F(\mathbf{x})$, т.е. не можем использовать правила Армихо и Вульфа.



SGD trajectory



optimum vicinity

- Тем не менее, можно попробовать проанализировать итерацию SGD для $F(\mathbf{x}) = \mathbb{E}_{q(\mathbf{y})} f(\mathbf{x}, \mathbf{y}) \rightarrow \min_{\mathbf{x}}$:

$$\mathbf{x}_{k+1} = \mathbf{x}_k - \alpha_k \hat{\mathbf{g}}_k, \quad \mathbb{E} \hat{\mathbf{g}}_k = \mathbf{g}_k = \nabla F(\mathbf{x}_k).$$

- Давайте оценим невязку точки на очередной итерации:

$$\begin{aligned} \|\mathbf{x}_{k+1} - \mathbf{x}_{\text{opt}}\|^2 &= \|\mathbf{x}_k - \alpha_k \hat{\mathbf{g}}_k - \mathbf{x}_{\text{opt}}\|^2 = \\ &= \|\mathbf{x}_k - \mathbf{x}_{\text{opt}}\|^2 - 2\alpha_k \hat{\mathbf{g}}_k^\top (\mathbf{x}_k - \mathbf{x}_{\text{opt}}) + \alpha_k^2 \|\hat{\mathbf{g}}_k\|^2. \end{aligned}$$

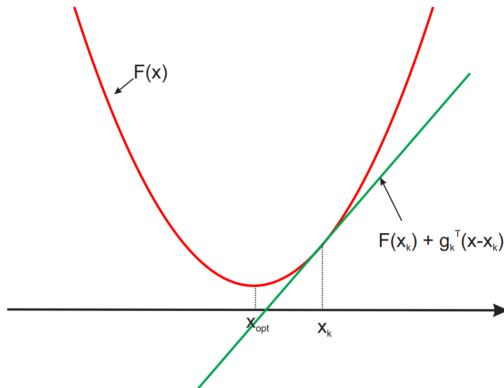
- Возьмём ожидание по $q(\mathbf{y})$ в момент времени k :

$$\mathbb{E} \|\mathbf{x}_{k+1} - \mathbf{x}_{\text{opt}}\|^2 = \|\mathbf{x}_k - \mathbf{x}_{\text{opt}}\|^2 - 2\alpha_k \mathbf{g}_k^\top (\mathbf{x}_k - \mathbf{x}_{\text{opt}}) + \alpha_k^2 \mathbb{E} \|\hat{\mathbf{g}}_k\|^2.$$

СТОХАСТИЧЕСКИЙ ГРАДИЕНТНЫЙ СПУСК

- Для простоты предположим, что F выпуклая:

$$F(\mathbf{x}_{\text{opt}}) \geq F(\mathbf{x}_k) + \mathbf{g}_k^\top (\mathbf{x}_k - \mathbf{x}_{\text{opt}})$$



- У нас было

$$\begin{aligned}\mathbb{E}\|\mathbf{x}_{k+1} - \mathbf{x}_{\text{opt}}\|^2 &= \|\mathbf{x}_k - \mathbf{x}_{\text{opt}}\|^2 - 2\alpha_k \mathbf{g}_k^\top (\mathbf{x}_k - \mathbf{x}_{\text{opt}}) + \alpha_k^2 \mathbb{E}\|\hat{\mathbf{g}}_k\|^2, \\ F(\mathbf{x}_{\text{opt}}) &\geq F(\mathbf{x}_k) + \mathbf{g}_k^\top (\mathbf{x}_k - \mathbf{x}_{\text{opt}}).\end{aligned}$$

- Значит,

$$\begin{aligned}\alpha_k (F(\mathbf{x}_k) - F(\mathbf{x}_{\text{opt}})) &\leq \alpha_k \mathbf{g}_k^\top (\mathbf{x}_k - \mathbf{x}_{\text{opt}}) = \\ &= \frac{1}{2} \|\mathbf{x}_k - \mathbf{x}_{\text{opt}}\|^2 + \frac{1}{2} \alpha_k^2 \mathbb{E}\|\hat{\mathbf{g}}_k\|^2 - \frac{1}{2} \mathbb{E}\|\mathbf{x}_{k+1} - \mathbf{x}_{\text{opt}}\|^2.\end{aligned}$$

- Возьмём ожидание от левой части и просуммируем:

$$\begin{aligned} \sum_{i=0}^k \alpha_i (\mathbb{E} F(\mathbf{x}_i) - F(\mathbf{x}_{\text{opt}})) &\leq \\ &\leq \frac{1}{2} \|\mathbf{x}_0 - \mathbf{x}_{\text{opt}}\|^2 + \frac{1}{2} \sum_{i=0}^k \alpha_i^2 \mathbb{E} \|\hat{\mathbf{g}}_i\|^2 - \frac{1}{2} \mathbb{E} \|\mathbf{x}_{k+1} - \mathbf{x}_{\text{opt}}\|^2 \leq \\ &\leq \frac{1}{2} \|\mathbf{x}_0 - \mathbf{x}_{\text{opt}}\|^2 + \frac{1}{2} \sum_{i=0}^k \alpha_i^2 \mathbb{E} \|\hat{\mathbf{g}}_i\|^2. \end{aligned}$$

- Получилась сумма значений функции в разных точках с весами α_i . Что делать?

- Воспользуемся выпуклостью:

$$\begin{aligned} \mathbb{E} F \left(\frac{\sum_i \alpha_i \mathbf{x}_i}{\sum_i \alpha_i} \right) - F(\mathbf{x}_{\text{opt}}) &\leq \\ &\leq \frac{\sum_i \alpha_i (\mathbb{E} F(\mathbf{x}_i) - F(\mathbf{x}_{\text{opt}}))}{\sum_i \alpha_i} \leq \frac{\frac{1}{2} \|\mathbf{x}_0 - \mathbf{x}_{\text{opt}}\|^2 + \frac{1}{2} \sum_{i=0}^k \alpha_i^2 \mathbb{E} \|\hat{\mathbf{g}}_i\|^2}{\sum_i \alpha_i}. \end{aligned}$$

- Т.е. оценка получилась на значение в линейной комбинации точек (поэтому в статьях часто берут среднее/ожидание или линейную комбинацию, а на практике нет разницы или лучше брать последнюю точку)
- Если $\|\mathbf{x}_0 - \mathbf{x}_{\text{opt}}\| \leq R$ и $\mathbb{E} \|\hat{\mathbf{g}}_k\|^2 \leq G^2$, то

$$\mathbb{E} F(\hat{\mathbf{x}}_k) - F(\mathbf{x}_{\text{opt}}) \leq \frac{R^2 + G^2 \sum_{i=0}^k \alpha_i^2}{2 \sum_{i=0}^k \alpha_i}.$$

- Это самая главная оценка про SGD:

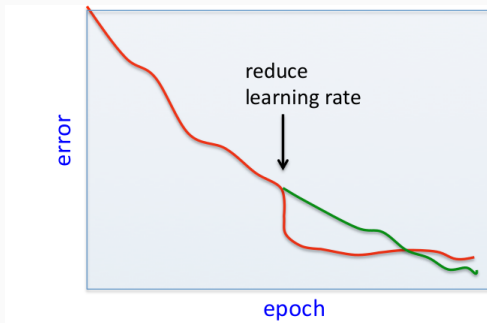
$$\mathbb{E}F(\hat{\mathbf{x}}_k) - F(\mathbf{x}_{\text{opt}}) \leq \frac{R^2 + G^2 \sum_{i=0}^k \alpha_i^2}{2 \sum_{i=0}^k \alpha_i}.$$

- R – оценка начальной невязки, а G – оценка чего-то вроде дисперсии стохастического градиента.
- Например, для постоянного шага $\alpha_i = h$

$$\mathbb{E}F(\hat{\mathbf{x}}_k) - F(\mathbf{x}_{\text{opt}}) \leq \frac{R^2}{2h(k+1)} + \frac{G^2 h}{2} \xrightarrow{k \rightarrow \infty} \frac{G^2 h}{2}.$$

- Итоги про SGD:
 - SGD приходит в «регион неопределённости» радиуса $\frac{1}{2}G^2h$, и этот радиус пропорционален длине шага;
 - чем быстрее идём, тем быстрее придём, но регион неопределённости будет больше, т.е. по идее надо уменьшать со временем скорость обучения;
 - SGD сходится медленно: полный GD для выпуклых функций сходится за $O(1/k)$, а SGD – за $O(1/\sqrt{k})$;
 - но далеко от региона неопределённости у нас скорость тоже $O(1/k)$ получилась для постоянной скорости обучения, т.е. замедляется только уже близко к оптимуму, и вообще цель наша – достичь региона неопределённости;
 - но всё равно всё зависит от G , и это будет особенно важно потом в нейробайесовских методах.

- Значит, нужны какие-то улучшения. Что-то делать со скоростью обучения.
- Скорость обучения лучше не уменьшать слишком быстро.



- Но это в любом случае никак не учитывает собственно F .

- *Метод моментов* (momentum): сохраним часть скорости, как у материальной точки.
- С инерцией получается

$$u_t = \gamma u_{t-1} + \eta \nabla_{\mathbf{x}} F(\mathbf{x}),$$

$$\mathbf{x} = \mathbf{x} - u_t.$$

- И теперь мы сохраняем γu_{t-1} .

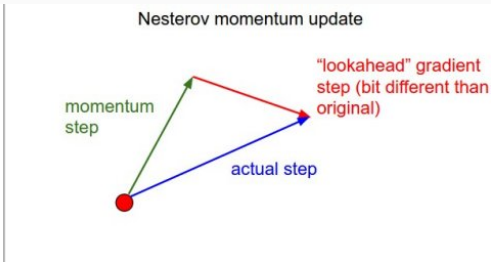
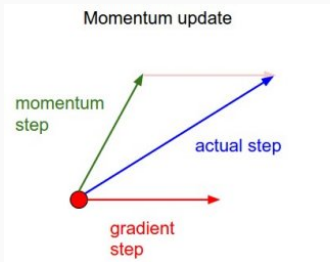


without momentum



with momentum

- Мы ведь на самом деле уже знаем, что попадём в γu_{t-1} на промежуточном шаге.
- Давайте прямо там, на полпути, и вычислим градиент!

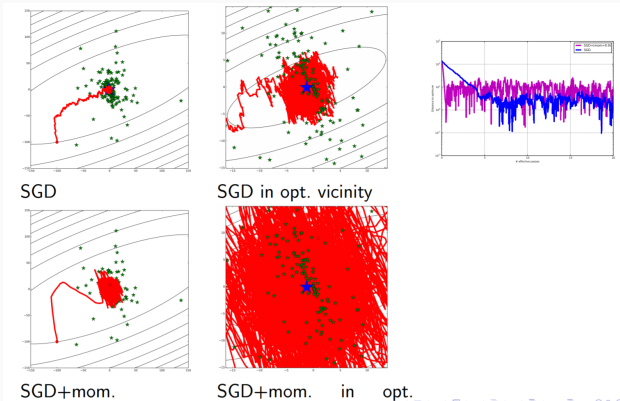


- *Метод Нестерова* (Nesterov's momentum):

$$u_t = \gamma u_{t-1} + \eta \nabla_{\mathbf{x}} F(\mathbf{x} - \gamma u_{t-1})$$



- Всё равно, конечно, проблемы не пропадают:



- Можно ли ещё лучше?..

- Заметим, что до сих пор скорость обучения была одна во всех направлениях, мы пытались выбрать направление как бы глобально.
- Идея: давайте быстрее двигаться по тем параметрам, которые не сильно меняются, и медленнее по быстро меняющимся параметрам.

- *Adagrad*: давайте накапливать историю этой скорости изменений и учитывать её.
- Обозначая $g_{t,i} = \nabla_{\mathbf{w}_i} L(\mathbf{w})$, получим

$$\mathbf{w}_{t+1,i} = \mathbf{w}_{t,i} - \frac{\eta}{\sqrt{G_{t,ii} + \epsilon}} \cdot g_{t,i},$$

где G_t – диагональная матрица с $G_{t,ii} = G_{t-1,ii} + g_{t,i}^2$, которая накапливает общее значение градиента по всей истории обучения.

- Так что скорость обучения всё время уменьшается, но с разной скоростью для разных $\mathbf{w}_i$.

- Проблема: G всё увеличивается и увеличивается, и скорость обучения иногда уменьшается слишком быстро.
- *Adadelta* (Zeiler, 2012) – та же идея, но две новых модификации.
- Во-первых, историю градиентов мы теперь считаем с затуханием:

$$G_{t,ii} = \rho G_{t-1,ii} + (1 - \rho) g_{t,i}^2.$$

- А всё остальное здесь точно так же:

$$\mathbf{u}_t = -\frac{\eta}{\sqrt{G_{t-1} + \epsilon}} \mathbf{g}_{t-1}.$$

- Во-вторых, надо бы «единицы измерения» привести в соответствие.
- В предыдущих методах была проблема:
 - в обычном градиентном спуске или методе моментов «единицы измерения» обновления параметров $\Delta \mathbf{w}$ — это единицы измерения градиента, т.е. если веса в секундах, а целевая функция в метрах, то градиент будет иметь размерность «метр в секунду», и мы вычитаем метры в секунду из секунд;
 - а в Adagrad получалось, что значения обновлений $\Delta \mathbf{w}$ зависели от отношений градиентов, и величина обновлений вовсе безразмерная.

- Эта проблема решается в методе второго порядка: обновление параметров $\Delta \mathbf{w}$ пропорционально $H^{-1} \nabla_{\mathbf{w}} f$, то есть размерность будет

$$\Delta \mathbf{w} \propto H^{-1} \nabla_{\mathbf{w}} f \propto \frac{\frac{\partial f}{\partial \mathbf{w}}}{\frac{\partial^2 f}{\partial \mathbf{w}^2}} \propto \text{размерность } \mathbf{w}.$$

- Чтобы привести *Adadelta* в соответствие, нужно домножить на ещё одно экспоненциальное среднее, но теперь уже от квадратов обновлений параметров, а не от градиента.
- Настоящее среднее мы не знаем, аппроксимируем предыдущими шагами:

$$\mathbb{E} [\Delta \mathbf{w}^2]_t = \rho \mathbb{E} [\Delta \mathbf{w}^2]_{t-1} + (1 - \rho) \Delta \mathbf{w}^2, \text{ где}$$
$$u_t = - \frac{\sqrt{\mathbb{E} [\Delta \mathbf{w}^2]_{t-1} + \epsilon}}{\sqrt{G_{t-1} + \epsilon}} \cdot g_{t-1}.$$

- Следующий вариант – *RMSprop* из курса Хинтона.
- Практически то же, что *Adadelta*, только *RMSprop* не делает вторую поправку с изменением единиц и хранением истории самих обновлений, а просто использует корень из среднего от квадратов (вот он где, RMS) от градиентов:

$$u_t = -\frac{\eta}{\sqrt{G_{t-1} + \epsilon}} \cdot g_{t-1}.$$

- И последний алгоритм – *Adam* (Kingma, Ba, 2014).
- Модификация *Adagrad* со сглаженными версиями среднего и среднеквадратичного градиентов:

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

$$v_t = \beta_2 v_{t-1} + (1 - \beta_2) g_t^2,$$

$$u_t = \frac{\eta}{\sqrt{v_t} + \epsilon} m_t.$$

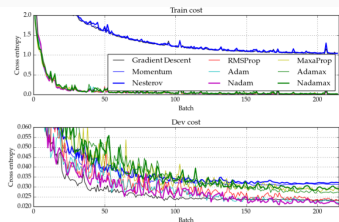
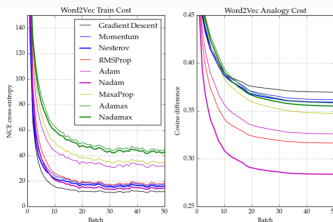
- (Kingma, Ba, 2014) рекомендуют $\beta_1 = 0.9$, $\beta_2 = 0.999$, $\epsilon = 10^{-8}$.
- *Adam* практически не требует настройки, используется на практике очень часто.
- ...[<http://ruder.io/optimizing-gradient-descent/>]...

АДАПТИВНЫЕ МЕТОДЫ ГРАДИЕНТНОГО СПУСКА

- А ещё можно совместить Adam и Нестерова – Nadam (Dozat, 2016)

Algorithm 8 Nesterov-accelerated adaptive moment estimation

$$\begin{aligned} \mathbf{g}_t &\leftarrow \nabla_{\theta_{t-1}} f(\theta_{t-1}) \\ \hat{\mathbf{g}} &\leftarrow \frac{\mathbf{g}_t}{1 - \prod_{i=1}^t \mu_i} \\ \mathbf{m}_t &\leftarrow \mu \mathbf{m}_{t-1} + (1 - \mu) \mathbf{g}_t \\ \hat{\mathbf{m}}_t &\leftarrow \frac{\mathbf{m}_t}{1 - \prod_{i=1}^t \mu_i} \\ \mathbf{n}_t &\leftarrow \nu \mathbf{n}_{t-1} + (1 - \nu) \mathbf{g}_t^2 \\ \hat{\mathbf{n}}_t &\leftarrow \frac{\mathbf{n}_t}{1 - \prod_{i=1}^t \nu_i} \\ \bar{\mathbf{m}}_t &\leftarrow (1 - \mu_t) \hat{\mathbf{g}}_t + \mu_{t+1} \hat{\mathbf{m}}_t \\ \theta_t &\leftarrow \theta_{t-1} - \eta \frac{\bar{\mathbf{m}}_t}{\sqrt{\hat{\mathbf{n}}_t} + \epsilon} \end{aligned}$$



- Чуть более общий взгляд – всё это выглядит вот так:

- обычный стохастический градиентный спуск:

$$\mathbf{w}_{k+1} = \mathbf{w}_k - \alpha_k \tilde{\nabla} f(\mathbf{w}_k), \text{ где } \tilde{\nabla} f(\mathbf{w}_k) = \nabla f(\mathbf{w}_k; \mathbf{x}_{i_k});$$

- SGD с моментами:

$$\mathbf{w}_{k+1} = \mathbf{w}_k - \alpha_k \tilde{\nabla} f(\mathbf{w}_k + \gamma_k (\mathbf{w}_k - \mathbf{w}_{k-1})) + \beta_k (\mathbf{w}_k - \mathbf{w}_{k-1});$$

- адаптивный SGD:

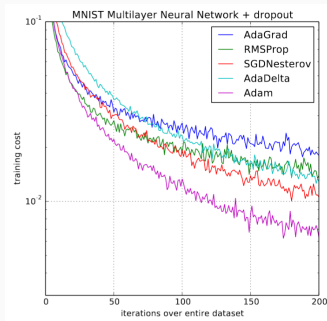
$$\mathbf{w}_{k+1} = \mathbf{w}_k - \alpha_k H_k^{-1} \tilde{\nabla} f(\mathbf{w}_k + \gamma_k (\mathbf{w}_k - \mathbf{w}_{k-1})) + \beta_k H_k^{-1} H_{k-1} (\mathbf{w}_k - \mathbf{w}_{k-1}),$$

где обычно $H_k = \text{diag} \left(\left[\sum_{i=1}^k \eta_i \mathbf{g}_i \circ \mathbf{g}_i \right]^{1/2} \right)$, где

$\mathbf{g}_k = \tilde{\nabla} f(\mathbf{w}_k + \gamma_k (\mathbf{w}_k - \mathbf{w}_{k-1}))$ (т.е. H_k – это диагональная матрица, элементы которой – квадратные корни из линейных комбинаций квадратов предыдущих градиентов).

- Т.е. адаптивные методы пытаются подстроиться под геометрию в пространстве данных, а SGD и его варианты используют базовую L_2 -геометрию с $H_k = \mathbf{I}$.

- Когда Adam появился, все были очень счастливы, и было отчего:

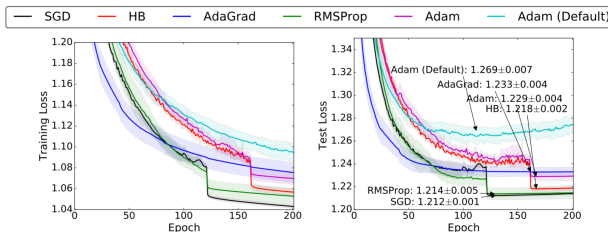
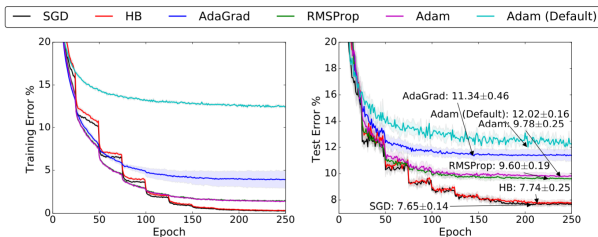


- Но потом выяснилось, что не всё так просто... часто применяли простой SGD. Почему?

- The Marginal Value of Adaptive Gradient Methods in Machine Learning (Wilson et al., May 2017)
- Основные выводы:
 - когда в задаче несколько глобальных минимумов, разные алгоритмы могут найти совершенно разные решения из одной и той же начальной точки;
 - в частности, адаптивные методы могут приходить к оверфиттингу, т.е. не-обобщающимся локальным решениям;
 - и это *не только теоретический худший случай, но и практика.*

ADAM, ADAMW И ДРУГИЕ ЖИВОТНЫЕ

- The Marginal Value of Adaptive Gradient Methods in Machine Learning (Wilson et al., May 2017)

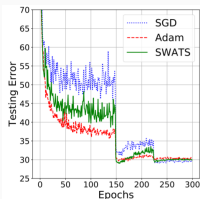


(a) War and Peace (Training Set)

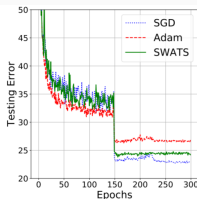
(b) War and Peace (Test Set)

ADAM, ADAMW и ДРУГИЕ ЖИВОТНЫЕ

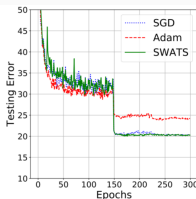
- Но Adam всё равно гораздо быстрее начинает, конечно.
- Поэтому предлагали, например, переключаться с Adam на SGD в нужное время (Keskar, Socher, Dec 2017)



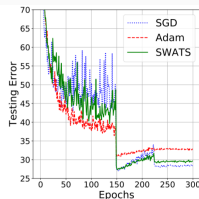
(e) ResNet-32 — CIFAR-100



(f) DenseNet — CIFAR-100



(g) PyramidNet — CIFAR-100



(h) SENet — CIFAR-100

- Всё равно, конечно, SGD не получилось превзойти, и даже не всегда наравне.
- Но и это ещё не вся история...

- Fixing Weight Decay Regularization in Adam (Loshchilov, Hutter, Feb 2018):
 - я мельком говорил, что weight decay – это то же самое, что L_2 -регуляризация;
 - но для адаптивных методов это, оказывается, не совсем так...
 - давайте разберёмся, что такое weight decay и почему это может быть не эквивалентно.

- Исходный weight decay (Hanson, Pratt, 1988):

$$\mathbf{x}_{t+1} = (1 - w)\mathbf{x}_t - \eta \nabla f_t(\mathbf{x}_t),$$

где w – это скорость weight decay, η – скорость обучения.

- Там же сразу отмечено, что это эквивалентно тому, чтобы поменять f :

$$f_t^{\text{reg}}(\mathbf{x}_t) = f_t(\mathbf{x}_t) + \frac{w}{2} \|\mathbf{x}_t\|_2^2.$$

- А можно и просто градиент подправить:

$$\nabla f_t^{\text{reg}}(\mathbf{x}_t) = \nabla f_t(\mathbf{x}_t) + w\mathbf{x}_t.$$

- Это всё, конечно, верно, но...

- ...но не работает уже даже просто с моментами:

Algorithm 1 **SGD with L_2 regularization** and
SGD with weight decay (SGDW), both with momentum

- 1: **given** initial learning rate $\alpha \in \mathbb{R}$, momentum factor $\beta_1 \in \mathbb{R}$, weight decay / L_2 regularization factor $w \in \mathbb{R}$
- 2: **initialize** time step $t \leftarrow 0$, parameter vector $\mathbf{x}_{t=0} \in \mathbb{R}^n$, first moment vector $\mathbf{m}_{t=0} \leftarrow \mathbf{0}$, schedule multiplier $\eta_{t=0} \in \mathbb{R}$
- 3: **repeat**
- 4: $t \leftarrow t + 1$
- 5: $\nabla f_t(\mathbf{x}_{t-1}) \leftarrow \text{SelectBatch}(\mathbf{x}_{t-1})$ $\triangleright$ select batch and return the corresponding gradient
- 6: $\mathbf{g}_t \leftarrow \nabla f_t(\mathbf{x}_{t-1}) + w\mathbf{x}_{t-1}$
- 7: $\eta_t \leftarrow \text{SetScheduleMultiplier}(t)$ $\triangleright$ can be fixed, decay, be used for warm restarts
- 8: $\mathbf{m}_t \leftarrow \beta_1\mathbf{m}_{t-1} + \eta_t\alpha\mathbf{g}_t$
- 9: $\mathbf{x}_t \leftarrow \mathbf{x}_{t-1} - \mathbf{m}_t - \eta_t w\mathbf{x}_{t-1}$
- 10: **until** *stopping criterion is met*
- 11: **return** optimized parameters $\mathbf{x}_t$

- Получается, что $\mathbf{x}_t$ затухает на $\alpha w\mathbf{x}_{t-1}$, а не на $w\mathbf{x}_{t-1}$; чтобы восстановить поведение, надо взять $w_t = \frac{\alpha}{\alpha'}\delta$, и теперь выбор гиперпараметров α и w завязан друг на друга.
- Решение – просто перенести decay в само изменение $\mathbf{x}_t$ (строка 9) и добавить масштабирование η_t .

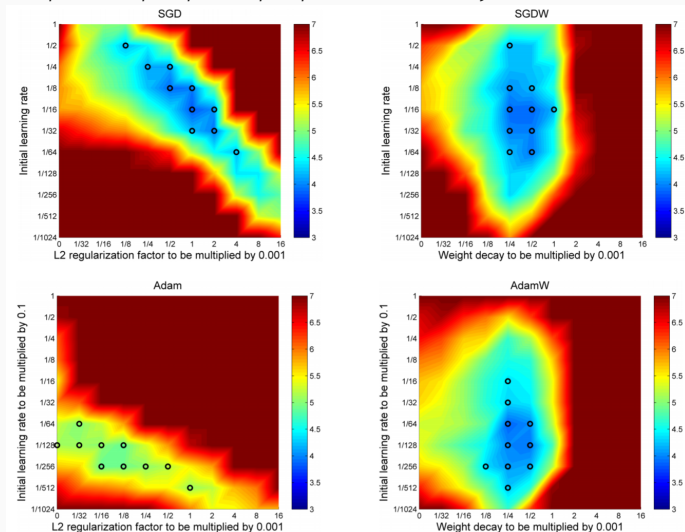
- То же самое происходит и с Adam:

Algorithm	2	Adam with L_2 regularization	and
Adam with weight decay (AdamW)			
1: given $\alpha = 0.001, \beta_1 = 0.9, \beta_2 = 0.999, \epsilon = 10^{-8}, w \in \mathbb{R}$ 2: initialize time step $t \leftarrow 0$, parameter vector $\mathbf{x}_{t=0} \in \mathbb{R}^n$, first moment vector $\mathbf{m}_{t=0} \leftarrow \mathbf{0}$, second moment vector $\mathbf{v}_{t=0} \leftarrow \mathbf{0}$, schedule multiplier $\eta_{t=0} \in \mathbb{R}$ 3: repeat 4: $t \leftarrow t + 1$ 5: $\nabla f_t(\mathbf{x}_{t-1}) \leftarrow \text{SelectBatch}(\mathbf{x}_{t-1})$ $\triangleright$ select batch and return the corresponding gradient 6: $\mathbf{g}_t \leftarrow \nabla f_t(\mathbf{x}_{t-1}) + w\mathbf{x}_{t-1}$ 7: $\mathbf{m}_t \leftarrow \beta_1\mathbf{m}_{t-1} + (1 - \beta_1)\mathbf{g}_t$ $\triangleright$ here and below all operations are element-wise 8: $\mathbf{v}_t \leftarrow \beta_2\mathbf{v}_{t-1} + (1 - \beta_2)\mathbf{g}_t^2$ 9: $\hat{\mathbf{m}}_t \leftarrow \mathbf{m}_t / (1 - \beta_1^t)$ $\triangleright \beta_1$ is taken to the power of t 10: $\hat{\mathbf{v}}_t \leftarrow \mathbf{v}_t / (1 - \beta_2^t)$ $\triangleright \beta_2$ is taken to the power of t 11: $\eta_t \leftarrow \text{SetScheduleMultiplier}(t)$ $\triangleright$ can be fixed, decay, or also be used for warm restarts 12: $\mathbf{x}_t \leftarrow \mathbf{x}_{t-1} - \eta_t \left(\alpha \hat{\mathbf{m}}_t / (\sqrt{\hat{\mathbf{v}}_t} + \epsilon) + w\mathbf{x}_{t-1} \right)$ 13: until stopping criterion is met 14: return optimized parameters $\mathbf{x}_t$			

- В базовом Adam веса с большими градиентами меньше затухают, что не всегда хорошо; AdamW восстанавливает исходное поведение.

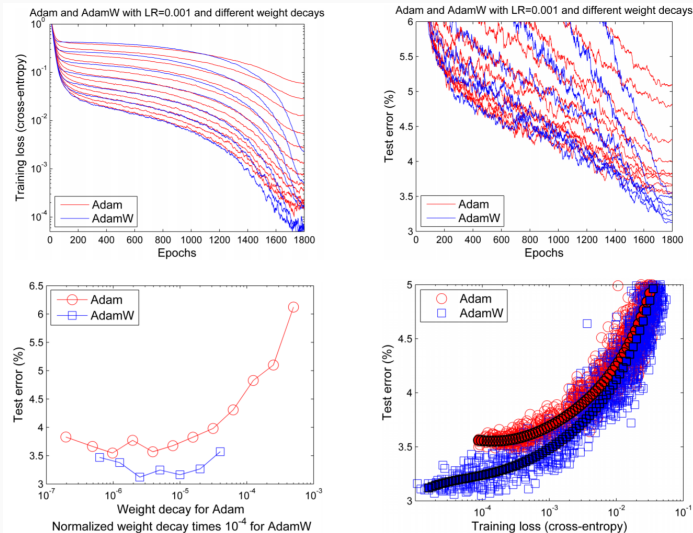
ADAM, ADAMW и ДРУГИЕ ЖИВОТНЫЕ

- И теперь гиперпараметры разделяются лучше:



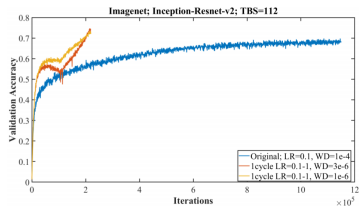
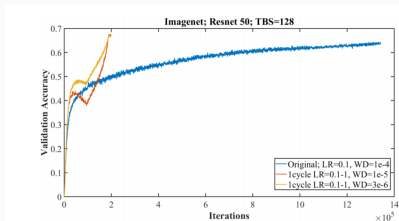
ADAM, ADAMW И ДРУГИЕ ЖИВОТНЫЕ

- Да и обобщается лучше:



SUPER-CONVERGENCE И AMSGRAD

- И ещё две мысли. Первая – Super-Convergence (Smith, Topin, Aug 2017)
- Идея в том, чтобы менять скорость обучения циклически –
- Это по сути смесь curriculum learning (Bengio et al., 2009) и классического simulated annealing.



- Но вроде бы это не воспроизводится устойчиво.

- Другая идея с похожей судьбой – amsgrad (Reddi et al., Mar 2018), ICLR 2018 best paper! Идея:
 - экспоненциальное среднее в Adam/RMSprop может привести к не-сходимости;
 - в частности, в доказательстве сходимости Adam есть ошибка;
 - а AMSGrad хранит максимальный размер градиента, и это типа лучше.

Algorithm 2 AMSGRAD

Input: $x_1 \in \mathcal{F}$, step size $\{\alpha_t\}_{t=1}^T$, $\{\beta_{1t}\}_{t=1}^T$, β_2
Set $m_0 = 0$, $v_0 = 0$ and $\hat{v}_0 = 0$
for $t = 1$ **to** T **do**
 $g_t = \nabla f_t(x_t)$
 $m_t = \beta_{1t}m_{t-1} + (1 - \beta_{1t})g_t$
 $v_t = \beta_2v_{t-1} + (1 - \beta_2)g_t^2$
 $\hat{v}_t = \max(\hat{v}_{t-1}, v_t)$ and $\hat{V}_t = \text{diag}(\hat{v}_t)$
 $x_{t+1} = \Pi_{\mathcal{F}, \sqrt{\hat{V}_t}}(x_t - \alpha_t m_t / \sqrt{\hat{v}_t})$
end for

- Но это тоже совсем не подтверждается на практике...

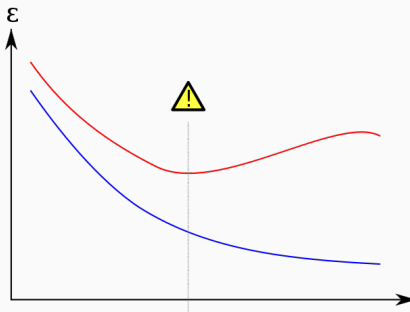
РЕГУЛЯРИЗАЦИЯ В НЕЙРОННЫХ СЕТЯХ

- У нейронных сетей *очень* много параметров.
- Регуляризация совершенно необходима.
- L_2 или L_1 регуляризация ($\lambda \sum_w w^2$ или $\lambda \sum_w |w|$) — это классический метод и в нейронных сетях, *weight decay*.
- Очень легко добавить: ещё одно слагаемое в целевую функцию, и иногда всё ещё полезно.

- Регуляризация есть во всех библиотеках. Например, в *Keras*:
 - `W_regularizer` добавит регуляризатор на матрицу весов слоя;
 - `b_regularizer` — на вектор свободных членов;
 - `activity_regularizer` — на вектор выходов.

РЕГУЛЯРИЗАЦИЯ В НЕЙРОННЫХ СЕТЯХ

- Второй способ регуляризации: *ранняя остановка* (early stopping).
- Давайте просто останавливаться, когда начнёт ухудшаться ошибка на валидационном множестве!
- Тоже есть из коробки в *Keras*, через callbacks.

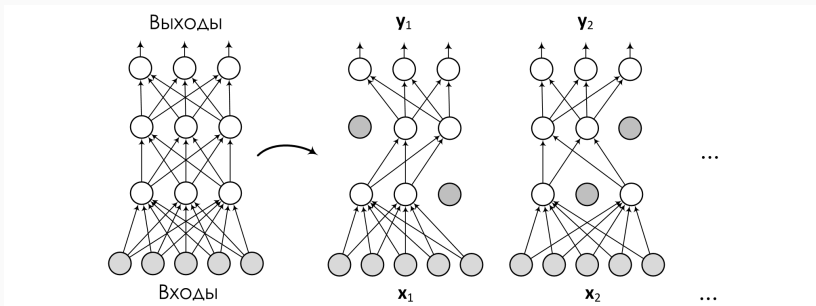


- Третий способ – max-norm constraint.
- Давайте искусственно ограничим норму вектора весов каждого нейрона:

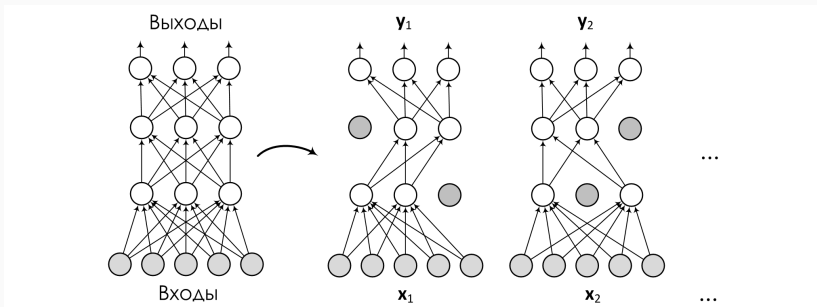
$$\|\mathbf{w}\|^2 \leq c.$$

- Это можно делать в процессе оптимизации: когда $\mathbf{w}$ выходит за шар радиуса c , проецируем его обратно.

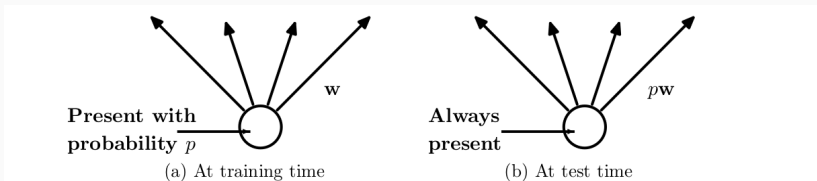
- Но есть и другие варианты.
- *Дропаут* (dropout): давайте просто выбросим некоторые нейроны случайным образом с вероятностью p !
(Srivastava et al., 2014)



- Получается, что мы сэмплируем кучу сетей, и нейрон получает на вход «среднюю» активацию от разных архитектур.
- Технический вопрос: как потом применять? Неужели надо опять сэмплировать кучу архитектур и усреднять?

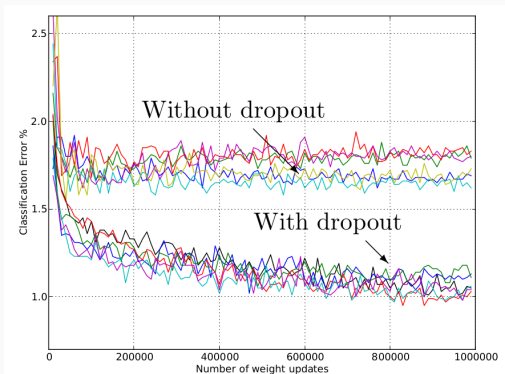


- Чтобы применить обученную сеть, умножим результат на $1/p$, сохраняя ожидание выхода!

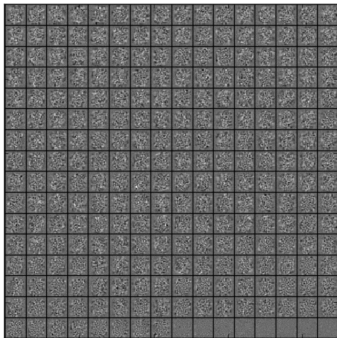


- В качестве вероятности часто можно брать просто $p = \frac{1}{2}$, большой разницы нет.

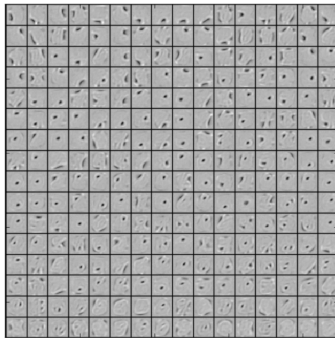
- Dropout улучшает (большие и свёрточные) нейронные сети *очень заметно*... но почему? WTF?



- Идея 1: нейроны теперь должны обучать признаки самостоятельно, а не рассчитывать на других.

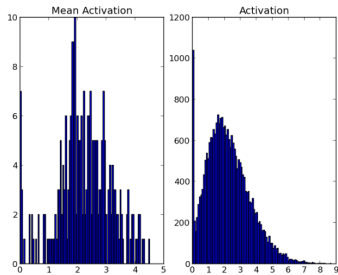


(a) Without dropout

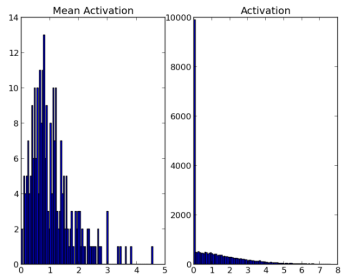


(b) Dropout with $p = 0.5$.

- Аналогично, и разреженность появляется.



(a) Without dropout



(b) Dropout with $p = 0.5$.

- Идея 2: мы как бы усредняем огромное число (2^N) сетей с общими весами, каждую обучая на один шаг.
- Усреднять кучу моделей очень полезно – bootstrapping/xgboost/всё такое...



- Получается, что дропаут – это такой экстремальный бутстреппинг.

- Идея 3: this is just like sex!
- Как работает половое размножение?
- Важно собрать не просто хорошую комбинацию, а *устойчивую* хорошую комбинацию.

BY ADI LIVNAT AND CHRISTOS PAPADIMITRIOU

Sex as an Algorithm

- Идея 4: нейрон посылает активацию a с вероятностью 0.5.
- Но можно наоборот: давайте посылать 0.5 (или 1) с вероятностью a .
- Ожидание то же, дисперсия для маленьких p растёт (что неплохо).
- И у нас получаются стохастические нейроны, которые посылают сигналы случайно – точно как в мозге!
- Улучшение примерно то же, как от dropout, но нужно меньше коммуникации между нейронами (один бит вместо float).
- Т.е. стохастические нейроны в мозге работают как дропаут-регуляризатор!
- Возможно, именно поэтому мы умеем задумываться.

- Идея 5: dropout — это специальная форма априорного распределения.
- Это очень полезный взгляд, с ним победили dropout в рекуррентных сетях.
- Но для этого нужно сначала поговорить о нейронных сетях по-байесовски...
- Вернёмся к этому, если будет время.

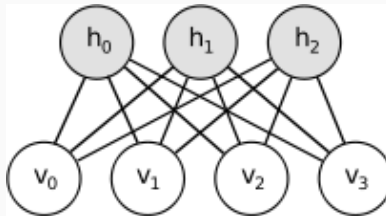
- Итого получается, что dropout – это очень крутой метод.



- Но и он сейчас отходит на второй план из-за нормализации по мини-батчам и новых версий градиентного спуска.
- О них чуть позже, а сначала об инициализации весов.

ИНИЦИАЛИЗАЦИЯ ВЕСОВ

- Революция глубокого обучения началась с *предобучением без учителя* (unsupervised pretraining).
- Главная идея: добраться до хорошей области пространства весов, затем уже сделать fine-tuning градиентным спуском.
- Ограниченные машины Больцмана (restricted Boltzmann machines):



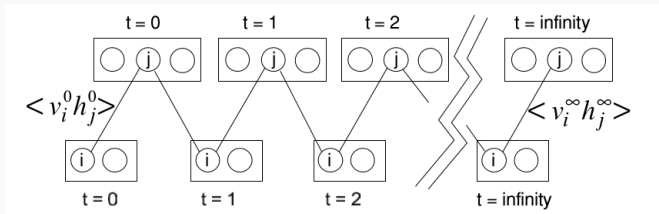
ПРЕДОБУЧЕНИЕ БЕЗ УЧИТЕЛЯ

- Это ненаправленная графическая модель, задающая распределение

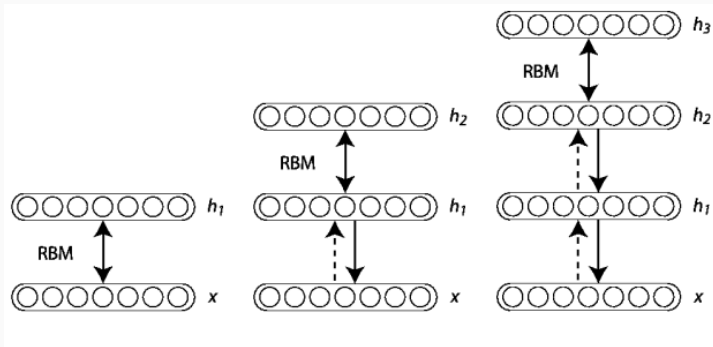
$$p(\mathbf{v}) = \sum_{\mathbf{h}} p(\mathbf{v}, \mathbf{h}) = \frac{1}{Z} \sum_{\mathbf{h}} e^{-E(\mathbf{v}, \mathbf{h})}, \text{ где}$$

$$E(\mathbf{v}, \mathbf{h}) = -\mathbf{b}^\top \mathbf{v} - \mathbf{c}^\top \mathbf{h} - \mathbf{h}^\top W \mathbf{v}.$$

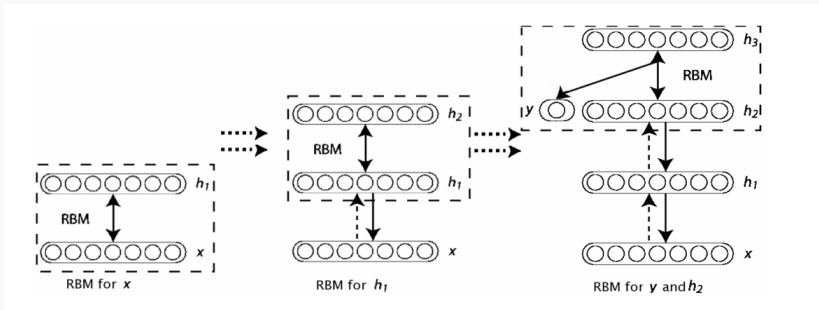
- Обучают алгоритмом Contrastive Divergence (приближение к сэмплированию по Гиббсу).



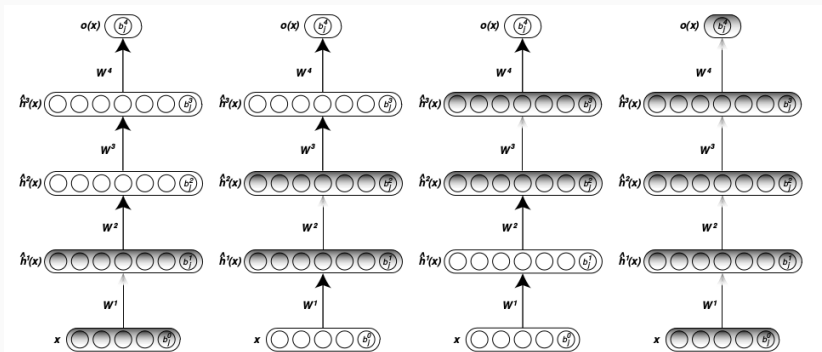
- Из RBM можно сделать глубокие сети, поставив одну на другую:



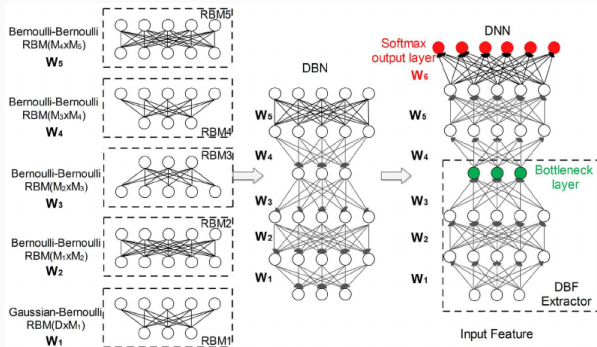
- И вывод можно вести последовательно, уровень за уровнем:



- А потом уже дообучать градиентным спуском (fine-tuning).



- Этот подход привёл к прорыву в распознавании речи.



- Но обучать глубокие сети из RBM довольно сложно, они хрупкие, и вычислительно тоже нелегко.
- И сейчас уже не очень-то и нужны сложные модели вроде RBM для того, чтобы попасть в хорошую начальную область.
- Инициализация весов — важная часть этого.

- *Xavier initialization* (Glorot, Bengio, 2010).
- Рассмотрим простой линейный нейрон:

$$y = \mathbf{w}^\top \mathbf{x} + b = \sum_i w_i x_i + b.$$

- Его дисперсия равна

$$\begin{aligned}\mathrm{Var}[y_i] &= \mathrm{Var}[w_i x_i] = \mathbb{E}[w_i^2 x_i^2] - (\mathbb{E}[w_i x_i])^2 = \\ &= \mathbb{E}[x_i]^2 \mathrm{Var}[w_i] + \mathbb{E}[w_i]^2 \mathrm{Var}[x_i] + \mathrm{Var}[w_i] \mathrm{Var}[x_i].\end{aligned}$$

- Его дисперсия равна

$$\begin{aligned}\text{Var}[y_i] &= \text{Var}[w_i x_i] = \mathbb{E}[w_i^2 x_i^2] - (\mathbb{E}[w_i x_i])^2 = \\ &= \mathbb{E}[x_i]^2 \text{Var}[w_i] + \mathbb{E}[w_i]^2 \text{Var}[x_i] + \text{Var}[w_i] \text{Var}[x_i].\end{aligned}$$

- Для нулевого среднего весов

$$\text{Var}[y_i] = \text{Var}[w_i] \text{Var}[x_i].$$

- И если w_i и x_i инициализированы независимо из одного и того же распределения,

$$\text{Var}[y] = \text{Var}\left[\sum_{i=1}^{n_{\text{out}}} y_i\right] = \sum_{i=1}^{n_{\text{out}}} \text{Var}[w_i x_i] = n_{\text{out}} \text{Var}[w_i] \text{Var}[x_i].$$

- Иначе говоря, дисперсия на выходе пропорциональна дисперсии на входе с коэффициентом $n_{\text{out}} \text{Var}[w_i]$.

- До (Glorot, Bengio, 2010) стандартным способом инициализации было

$$w_i \sim U \left[-\frac{1}{\sqrt{n_{\text{out}}}}, \frac{1}{\sqrt{n_{\text{out}}}} \right].$$

- См., например, *Neural Networks: Tricks of the Trade*.
- Так что с дисперсиями получается

$$\text{Var}[w_i] = \frac{1}{12} \left(\frac{1}{\sqrt{n_{\text{out}}}} + \frac{1}{\sqrt{n_{\text{out}}}} \right)^2 = \frac{1}{3n_{\text{out}}}, \text{ и}$$

$$n_{\text{out}} \text{Var}[w_i] = \frac{1}{3},$$

и после нескольких уровней сигнал совсем умирает;
аналогичный эффект происходит и в backprop.

- Инициализация Ксавье — давайте попробуем уменьшить изменение дисперсии, т.е. взять

$$\text{Var}[w_i] = \frac{2}{n_{\text{in}} + n_{\text{out}}};$$

для равномерного распределения это

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

- Но это работает только для симметричных активаций, т.е. не для ReLU...

- ...до работы (He et al., 2015). Вернёмся к

$$\text{Var} [w_i x_i] = \mathbb{E} [x_i]^2 \text{Var} [w_i] + \mathbb{E} [w_i]^2 \text{Var} [x_i] + \text{Var} [w_i] \text{Var} [x_i]$$

- Мы теперь можем обнулить только второе слагаемое:

$$\text{Var} [w_i x_i] = \mathbb{E} [x_i]^2 \text{Var} [w_i] + \text{Var} [w_i] \text{Var} [x_i] = \text{Var} [w_i] \mathbb{E} [x_i^2], \text{ и}$$

$$\text{Var} [y^{(l)}] = n_{\text{in}}^{(l)} \text{Var} [w^{(l)}] \mathbb{E} \left[(x^{(l)})^2 \right].$$

- Мы теперь можем обнулить только второе слагаемое:

$$\text{Var} [y^{(l)}] = n_{\text{in}}^{(l)} \text{Var} [w^{(l)}] \mathbb{E} \left[(x^{(l)})^2 \right].$$

- Предположим, что $x^{(l)} = \max(0, y^{(l-1)})$, и у $y^{(l-1)}$ симметричное распределение вокруг нуля. Тогда

$$\mathbb{E} \left[(x^{(l)})^2 \right] = \frac{1}{2} \text{Var} [y^{(l-1)}], \quad \text{Var} [y^{(l)}] = \frac{n_{\text{in}}^{(l)}}{2} \text{Var} [w^{(l)}] \text{Var} [y^{(l-1)}].$$

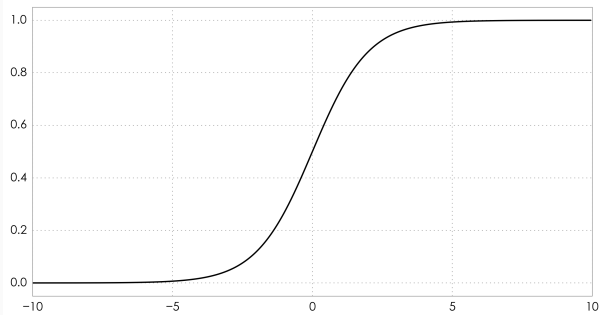
- И это приводит к формуле для дисперсии активации ReLU; теперь нет никакого n_{out} :

$$\text{Var}[w_i] = 2/n_{\text{in}}^{(l)}.$$

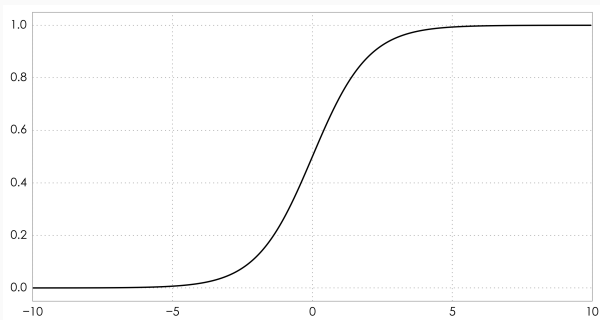
- Кстати, равномерную инициализацию делать не обязательно, можно и нормальное распределение:

$$w_i \sim N\left(0, \sqrt{2/n_{\text{in}}^{(l)}}\right).$$

- Кстати, о (Glorot, Bengio, 2010) – ещё одна важная идея.
- Эксперименты показали, что $\sigma(x) = \frac{1}{1+e^{-x}}$ работает в глубоких сетях довольно плохо.



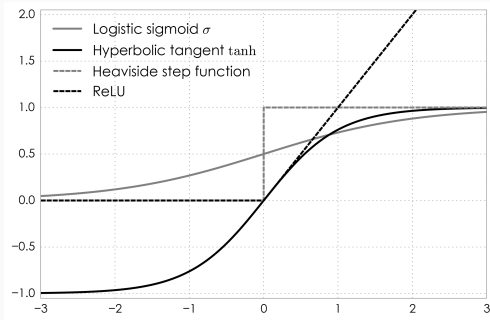
- *Насыщение*: если $\sigma(x)$ уже «обучилась», т.е. даёт большие по модулю значения, то её производная близка к нулю и «поменять мнение» трудно.



- Но ведь другие тоже насыщаются? В чём разница?

- Рассмотрим последний слой сети $h(W\mathbf{a} + \mathbf{b})$, где $\mathbf{a}$ — выходы предыдущего слоя, $\mathbf{b}$ — свободные члены, h — функция активации последнего уровня, обычно `softmax`.
- Когда мы начинаем оптимизировать сложную функцию потерь, поначалу выходы $\mathbf{h}$ не несут полезной информации о входах, ведь первые уровни ещё не обучены.
- Тогда неплохим приближением будет константная функция, выдающая средние значения выходов.
- Это значит, что $h(W\mathbf{a} + \mathbf{b})$ подберёт подходящие свободные члены $\mathbf{b}$ и постарается обнулить слагаемое $W\mathbf{h}$, которое поначалу скорее шум, чем сигнал.

- Иначе говоря, в процессе обучения мы постараемся привести выходы предыдущего слоя к нулю.
- Здесь и проявляется разница: у $\sigma(x) = \frac{1}{1+e^{-x}}$ область значений $(0, 1)$ при среднем $\frac{1}{2}$, и при $\sigma(x) \rightarrow 0$ будет и $\sigma'(x) \rightarrow 0$.
- А у $\tanh$ наоборот: когда $\tanh(x) \rightarrow 0$, $\tanh'(x)$ максимальна.

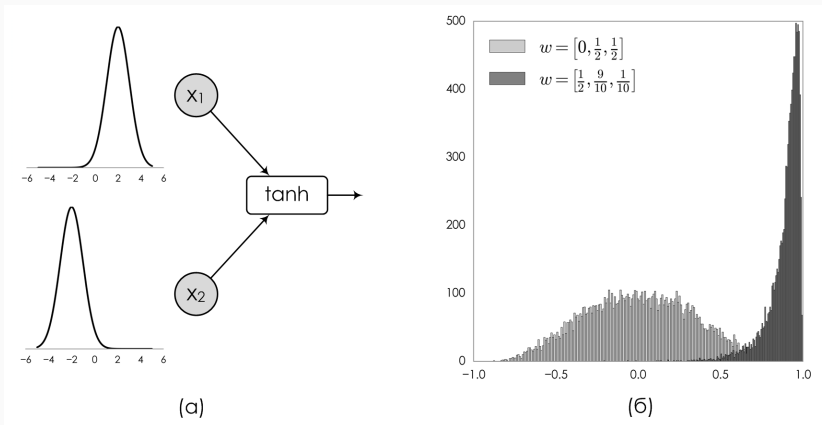


НОРМАЛИЗАЦИЯ ПО МИНИ-БАТЧАМ

- Ещё одна важная проблема в глубоких сетях: *внутренний сдвиг переменных* (internal covariate shift).
- Когда меняются веса слоя, меняется распределение его выходов.
- Это значит, что следующему уровню придётся всё начинать заново, он же не ожидал таких входов, не видел их раньше!
- Более того, нейроны следующего уровня могли уже и насытиться, и им теперь сложно быстро обучиться заново.
- Это серьёзно мешает обучению.

НОРМАЛИЗАЦИЯ ПО МИНИ-БАТЧАМ

- Вот характерный пример:



- Что делать?

- Можно пытаться нормализовать входы каждого уровня.
- Не работает: рассмотрим для простоты уровень с одним только bias b и входами $\mathbf{u}$:

$$\hat{\mathbf{x}} = \mathbf{x} - \mathbb{E}[\mathbf{x}], \text{ где } \mathbf{x} = \mathbf{u} + b.$$

- На следующем шаге градиентного спуска получится $b := b + \Delta b...$
- ...но $\hat{\mathbf{x}}$ не изменится:

$$\mathbf{u} + b + \Delta b - \mathbb{E}[\mathbf{u} + b + \Delta b] = \mathbf{u} + b - \mathbb{E}[\mathbf{u} + b].$$

- Так что всё обучение сведётся к тому, что b будет неограниченно расти — не очень хорошо.

- Можно пытаться добавить нормализацию как отдельный слой:

$$\hat{\mathbf{x}} = \text{Norm}(\mathbf{x}, X).$$

- Это лучше, но теперь этому слою на входе нужен весь датасет X !
- И на шаге градиентного спуска придётся вычислить $\frac{\partial \text{Norm}}{\partial \mathbf{x}}$ и $\frac{\partial \text{Norm}}{\partial X}$, да ещё и матрицу ковариаций

$$\text{Cov}[\mathbf{x}] = \mathbb{E}_{\mathbf{x} \in X} [\mathbf{x} \mathbf{x}^\top] - \mathbb{E}[\mathbf{x}] \mathbb{E}[\mathbf{x}]^\top.$$

- Это точно не работает.

- Решение в том, чтобы нормализовать каждый вход отдельно, и не по всему датасету, а по текущему кусочку; это и есть *нормализация по мини-батчам* (batch normalization).
- После нормализации по мини-батчам получим

$$\hat{x}_k = \frac{x_k - \mathbb{E}[x_k]}{\sqrt{\text{Var}[x_k]}},$$

где статистики подсчитаны по текущему мини-батчу.

- Ещё одна проблема: теперь пропадают нелинейности!
- Например, σ теперь практически всегда близка к линейной.

- Чтобы это исправить, нужно добавить гибкости уровню batchnorm.
- В частности, нужно разрешить ему обучаться иногда *ничего не делать* со входами.
- Так что вводим дополнительные параметры (сдвиг и растяжение):

$$y_k = \gamma_k \hat{x}_k + \beta_k = \gamma_k \frac{x_k - \mathbb{E}[x_k]}{\sqrt{\text{Var}[x_k]}} + \beta_k.$$

- γ_k и β_k — это новые переменные, тоже будут обучаться градиентным спуском, как веса.

НОРМАЛИЗАЦИЯ ПО МИНИ-БАТЧАМ

- И ещё добавим ϵ в знаменатель, чтобы на ноль не делить.
- Теперь мы можем формально описать слой батч-нормализации — для очередного мини-батча $B = \{\mathbf{x}_1, \dots, \mathbf{x}_m\}$:

- вычислить базовые статистики по мини-батчу

$$\mu_B = \frac{1}{m} \sum_{i=1}^m \mathbf{x}_i, \quad \sigma_B^2 = \frac{1}{m} \sum_{i=1}^m (\mathbf{x}_i - \mu_B)^2,$$

- нормализовать входы

$$\hat{\mathbf{x}}_i = \frac{\mathbf{x}_i - \mu_b}{\sqrt{\sigma_B^2 + \epsilon}},$$

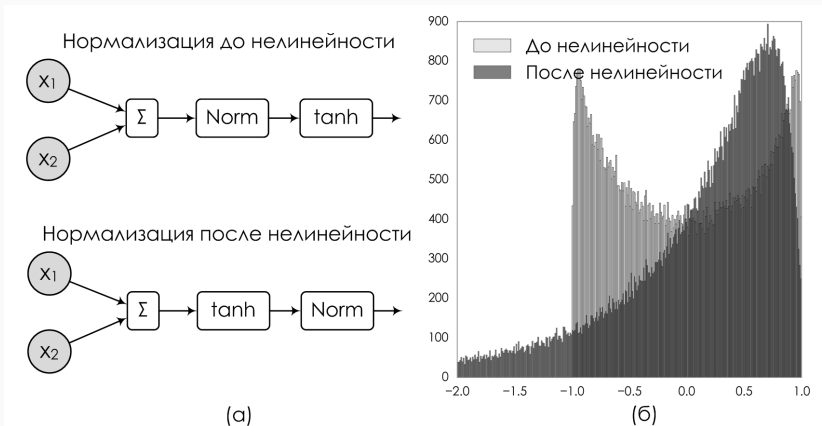
- вычислить результат

$$\mathbf{y}_i = \gamma \mathbf{x}_i + \beta.$$

- Через всё это совершенно стандартным образом пропускаются градиенты, в том числе по γ и β .

НОРМАЛИЗАЦИЯ ПО МИНИ-БАТЧАМ

- Последнее замечание: важно, куда поместить слой batchnorm.
- Можно до, а можно после нелинейности.



Что BN на самом деле делает

- Изначально идея была в том, чтобы сократить internal covariate shift.
- Но выяснилось, что от этого эффективность BN не особенно зависит, но BN помогает с регуляризацией, делает дропаут ненужным и т.д. Возможные причины:
 - BN сглаживает градиенты, уменьшает липшицеву константу (Santurkar et al., 2019);
 - BN разделяет обучение направления и длины векторов весов, что улучшает обучение;
 - есть результаты о том, что BN помогает добиться линейной сходимости в обычном градиентном спуске.

- Нормализация по мини-батчам сейчас стала фактически стандартом.
- Очередная очень крутая история, примерно как дропаут.
- Но мысль идёт и дальше:
 - (Laurent et al., 2016): BN не помогает рекуррентным сетям;
 - (Cooijmans et al., 2016): recurrent batch normalization;
 - (Salimans and Kingma, 2016): нормализация весов для улучшения обучения — давайте добавим веса как

$$\mathbf{h}_i = f \left(\frac{\gamma}{\|\mathbf{w}_i\|} \mathbf{w}_i^\top \mathbf{x} + b_i \right);$$

тогда мы будем перемасштабировать градиент, стабилизировать его норму и приближать его матрицу ковариаций к единичной, что улучшает обучение.

СПАСИБО!

Спасибо за внимание!

