КАК ОБУЧАТЬ НЕЙРОННЫЕ СЕТИ

Сергей Николенко

ИТМО — Санкт-Петербург

21 сентября 2026 г.

Random facts:

- 21 сентября 862 г. новгородцы призвали на княжение братьев-варягов Рюрика, Синеуса и Трувора
- 21 сентября 1348 г. евреи Цюриха обвинены в отравлении родников, а 21 сентября 1451 г. евреям Голландии было предписано носить на одежде опознавательные знаки
- 21 сентября 1745 г. состоялось первое сражение восстания якобитов: в битве при Престонпасе якобиты рассеяли английскую армию менее чем за двадцать минут
- 21 сентября 1937 г. в издательстве George Allen & Unwin вышла повесть Дж. Р. Р. Толкина «The Hobbit, or There and Back Again»
- 21 сентября в России — День коллекционера, отмечавшийся с конца 1950-х как День почтовой марки и коллекционера
- 21 сентября 1898 г. императрица Цыси захватила власть в Японии в результате дворцового переворота, закончив Сто дней реформ императора Айсиньгёро Цзайтяня

Краткое резюме предыдущей серии

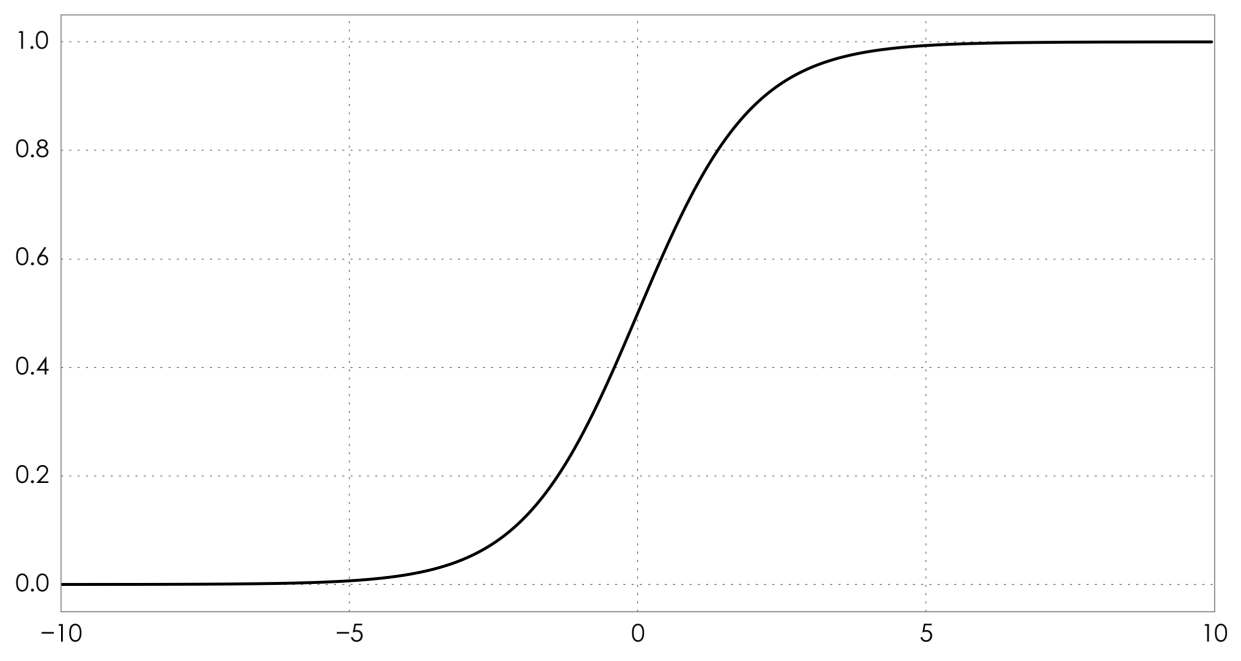
- В прошлый раз мы прошли путь от механических автоматов до перцептрона и вспомнили байесовский вывод: монетку, линейную и логистическую регрессию.
- Мы уже знаем, что такое один нейрон и что логистическая регрессия — это ровно он и есть.
- Сегодня разберёмся, как из нейронов собрать сеть и, главное, как её обучить:
  (1) функции активации — чем заменить сигмоиду и почему;
  (2) от перцептрона к сети, граф вычислений, backpropagation;
  (3) варианты градиентного спуска — от импульса до Adam;
  (4) и общие сюжеты: регуляризация и дропаут, инициализация весов, нормализация по мини-батчам.
- Материала много; что не успеем, перенесём на следующий раз.

Один перцептрон: функции активации

Функции активации

- Есть много разных нелинейных функций, которые можно использовать в перцептроне.
- Логистический сигмоид:

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

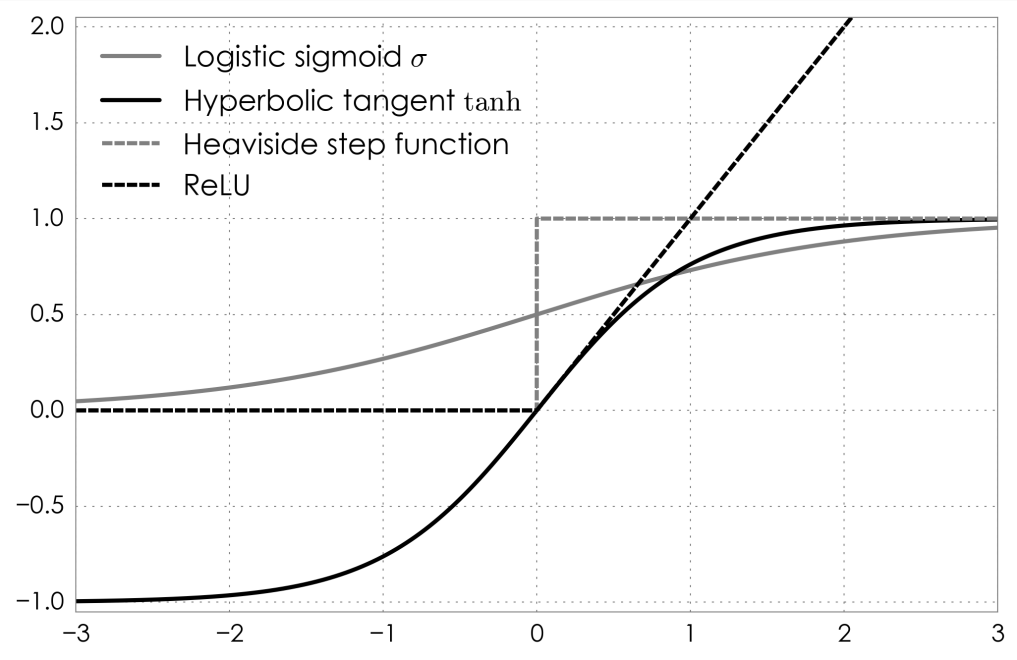


Функции активации

- Гиперболический тангенс:

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

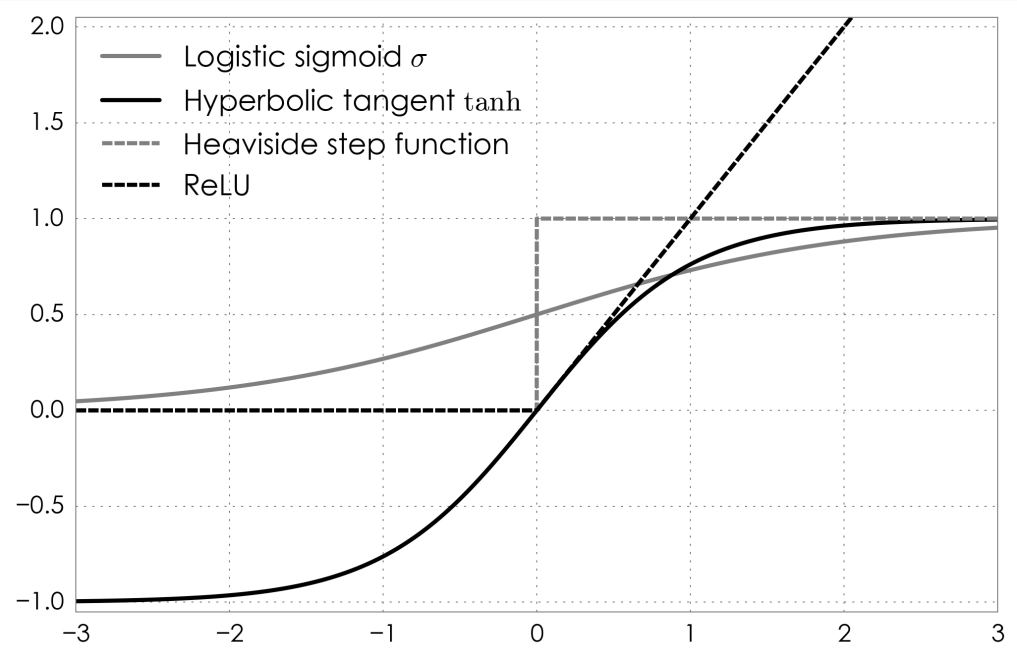
- Очень похоже, но ноль не является точкой насыщения.



Функции активации

- Ступенька (функция Хевисайда):

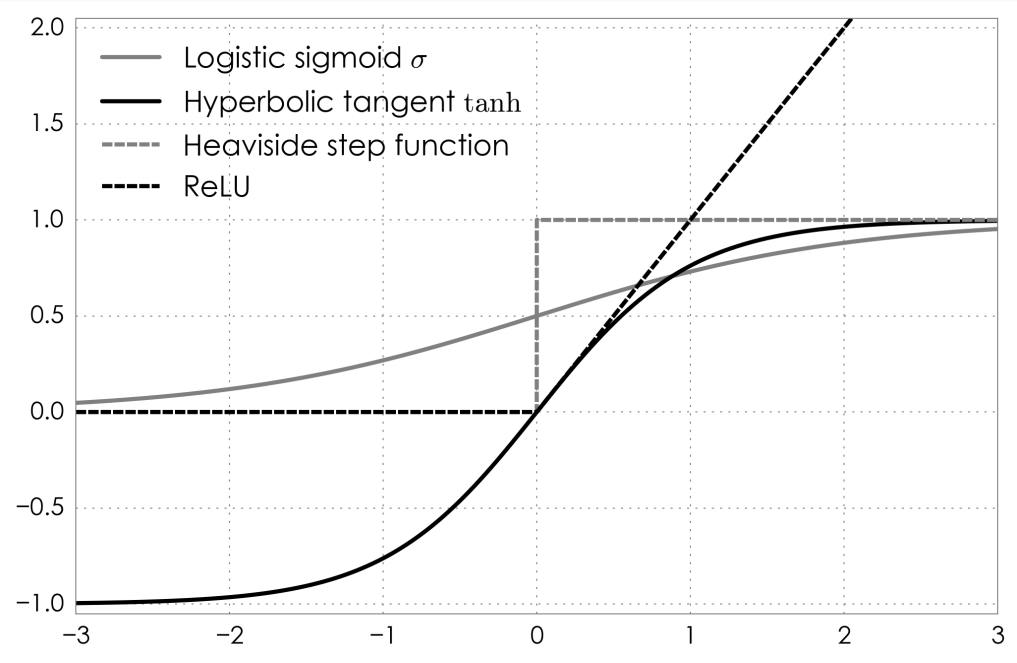
$$\text{step}(x) = \begin{cases} 0, & \text{if } x < 0, \\ 1, & \text{if } x > 0. \end{cases}$$



Функции активации

- Rectified linear unit (ReLU):

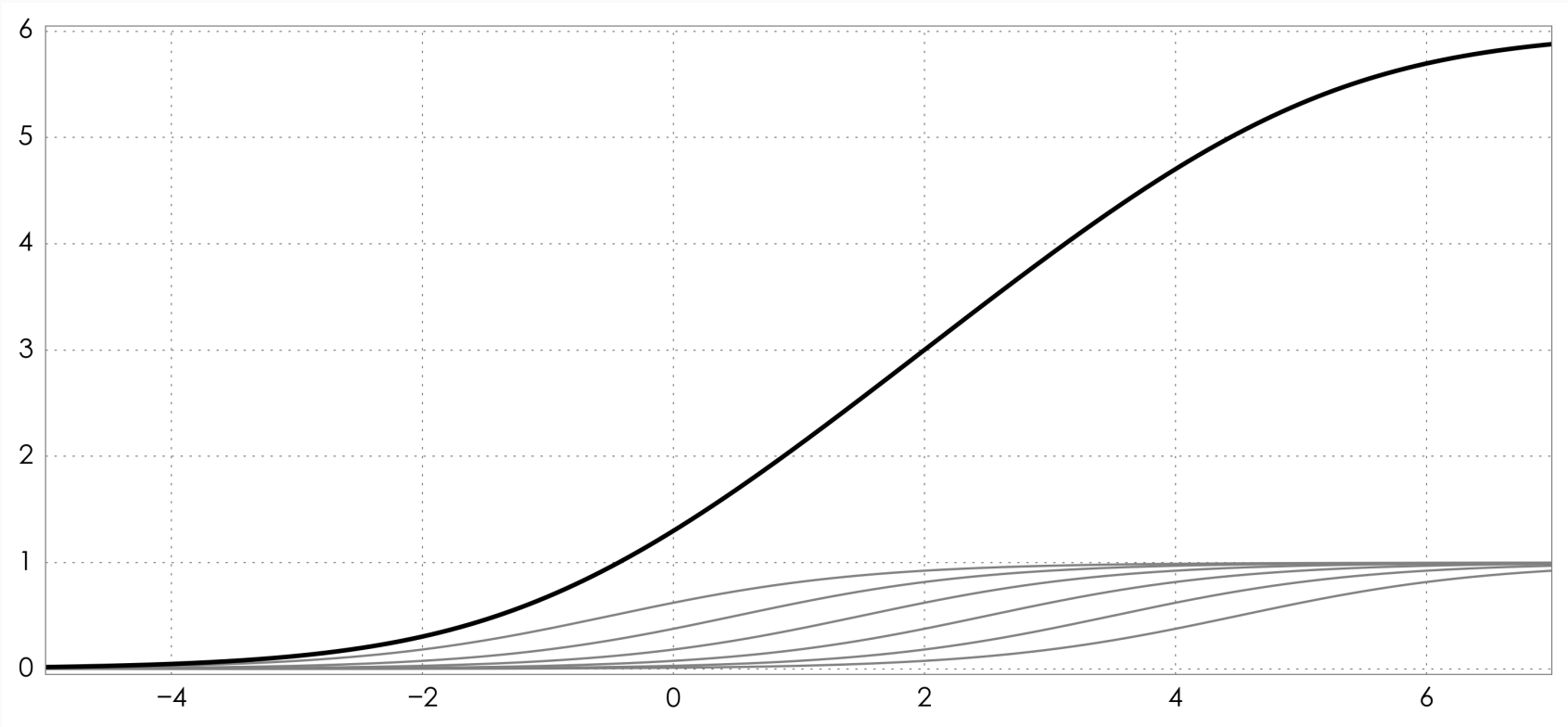
$$\mathrm{ReLU}(x) = \begin{cases} 0, & \text{if } x < 0, \\ x, & \text{if } x \geq 0. \end{cases}$$



Функции активации

- Математическая мотивация для ReLU:

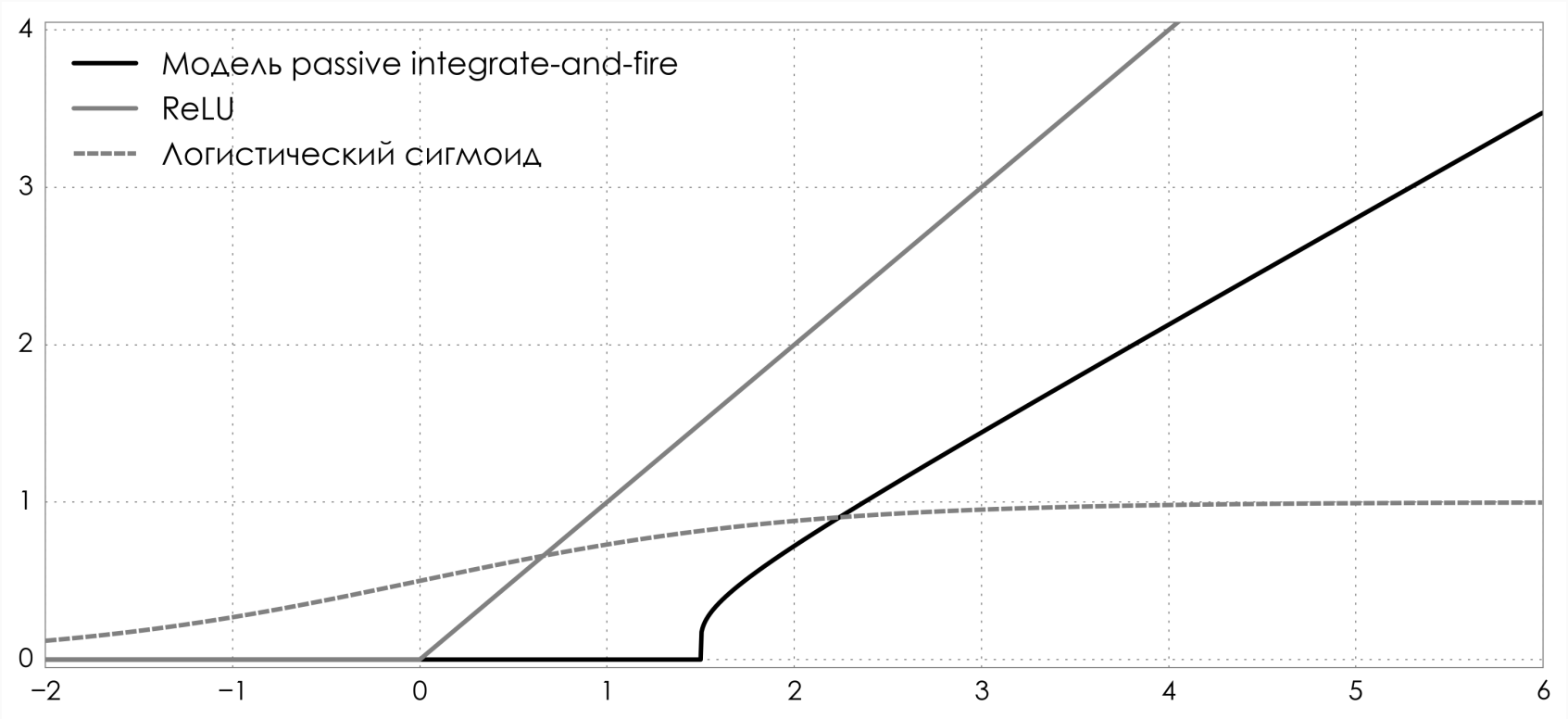
$$\sigma\left(x+\frac{1}{2}\right)+\sigma\left(x-\frac{1}{2}\right)+\sigma\left(x-\frac{3}{2}\right)+\sigma\left(x-\frac{5}{2}\right)+\ldots.$$



Функции активации

- Биологическая мотивация для ReLU:

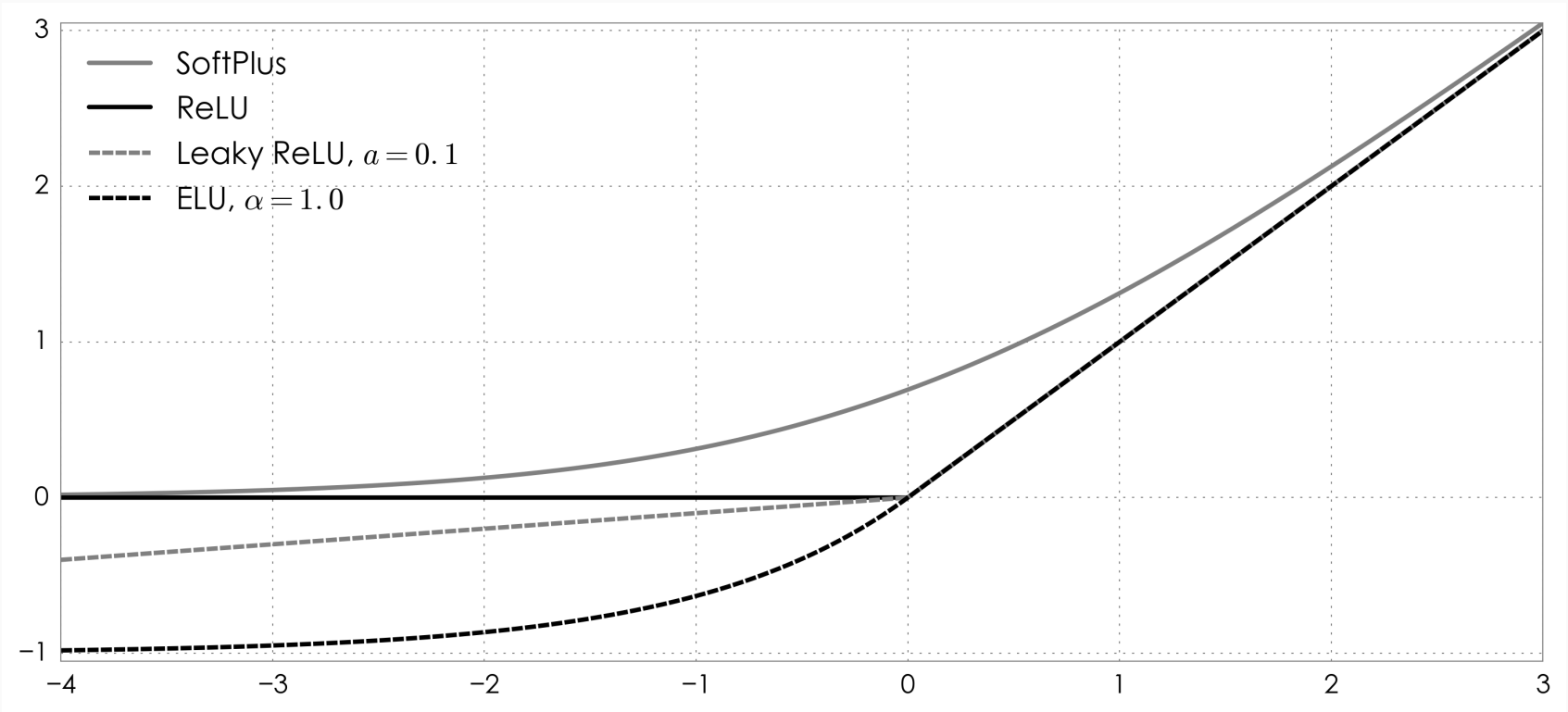
$$f(I) = \begin{cases} \left(\tau \log \frac{E+RI-V_{\text{reset}}}{E+RI-V_{th}}\right)^{-1}, & \text{if } E + RI > V_{th}, \\ 0, & \text{if } E + RI \leq V_{th}, \end{cases}$$



Функции активации

- Несколько вариантов ReLU — Leaky ReLU и Parametric ReLU:

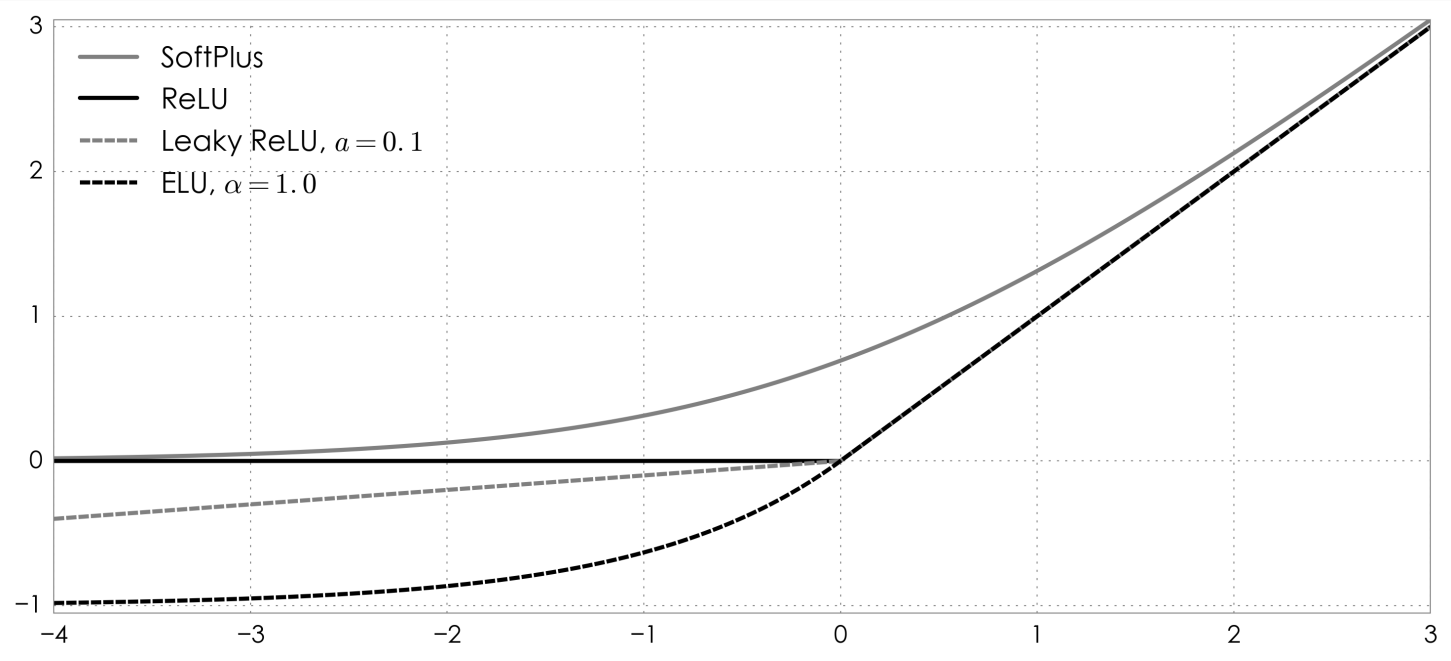
$$\mathrm{LReLU}(x) = \begin{cases} ax, & \text{if } x < 0, \\ x, & \text{if } x > 0. \end{cases}.$$



Функции активации

- Exponential LU:

$$\mathrm{ELU}(x) = \begin{cases} \alpha\left(e^x - 1\right), & x < 0, \\ x, & x \geq 0. \end{cases}$$



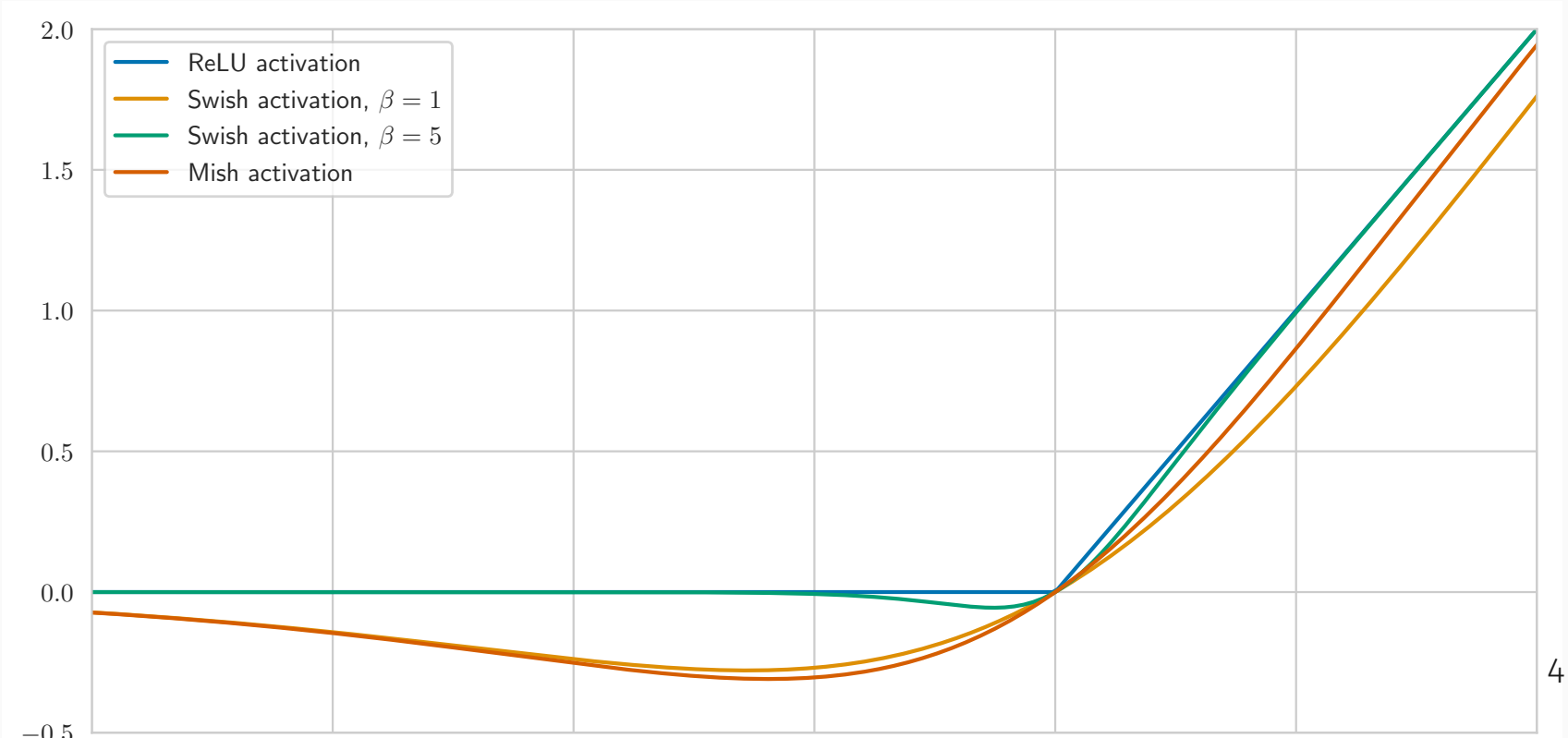
- Обычно ReLU хорошо работает, но последнего слова ещё не сказано.

Функции активации

- Но история только начинается. Ramachandran et al. (2017) сделали поиск по пространству функций и нашли *Swish*:

$$\text{Swish}(x) = x\sigma\left(\beta x\right) = \frac{x}{1 + e^{-\beta x}}$$

- (Misra, 2019): $\text{Mish}(x) = x \tanh\left(\text{softplus}(x)\right) = x \tanh\left(\ln\left(1 + e^{x}\right)\right)$



Функции активации

- Очень интересная работа вышла в сентябре 2020 года...
- (Ma et al., 2020): давайте рассмотрим гладкое обобщение максимума

$$S_\beta(x_1, \dots, x_n) = \frac{\sum_{i=1}^n x_i e^{\beta x_i}}{\sum_{i=1}^n e^{\beta x_i}}.$$

- И давайте подставим туда две функции от x, т.е. жёсткому максимуму $\max(g(x), h(x))$ будет соответствовать

$$\begin{aligned} S_\beta(g(x), h(x)) &= g(x)\frac{e^{\beta g(x)}}{e^{\beta g(x)} + e^{\beta h(x)}} + h(x)\frac{e^{\beta h(x)}}{e^{\beta g(x)} + e^{\beta h(x)}} = \\ &= g(x)\sigma(\beta(g(x) - h(x))) + h(x)\sigma(\beta(h(x) - g(x))) = \\ &= (g(x) - h(x))\,\sigma(\beta(g(x) - h(x))) + h(x). \end{aligned}$$

Функции активации

- Ma et al. назвали это *ActivateOrNot* (ACON). Теперь это обобщает всё на свете:
  - для $g(x) = x$ и $h(x) = 0$ жёсткий максимум будет $\mathrm{ReLU}(x) = \max(x, 0)$, а гладкий максимум

$$f_{ACON-A}(x, 0) = S_\beta(x, 0) = x\sigma(\beta x),$$

    т.е. в точности *Swish*!
  - для $g(x) = x$ и $h(x) = ax$ при некотором $a < 1$, жёсткий максимум будет $\mathrm{LReLU}(x) = \max(x, px)$, а гладкий максимум

$$f_{ACON-B} = S_\beta(x, ax) = (1 - a)\, x\sigma(\beta(1 - a)x) + ax;$$

Функции активации

- Ma et al. назвали это *ActivateOrNot* (ACON). Теперь это обобщает всё на свете:
  - и это можно прямолинейно обобщить до

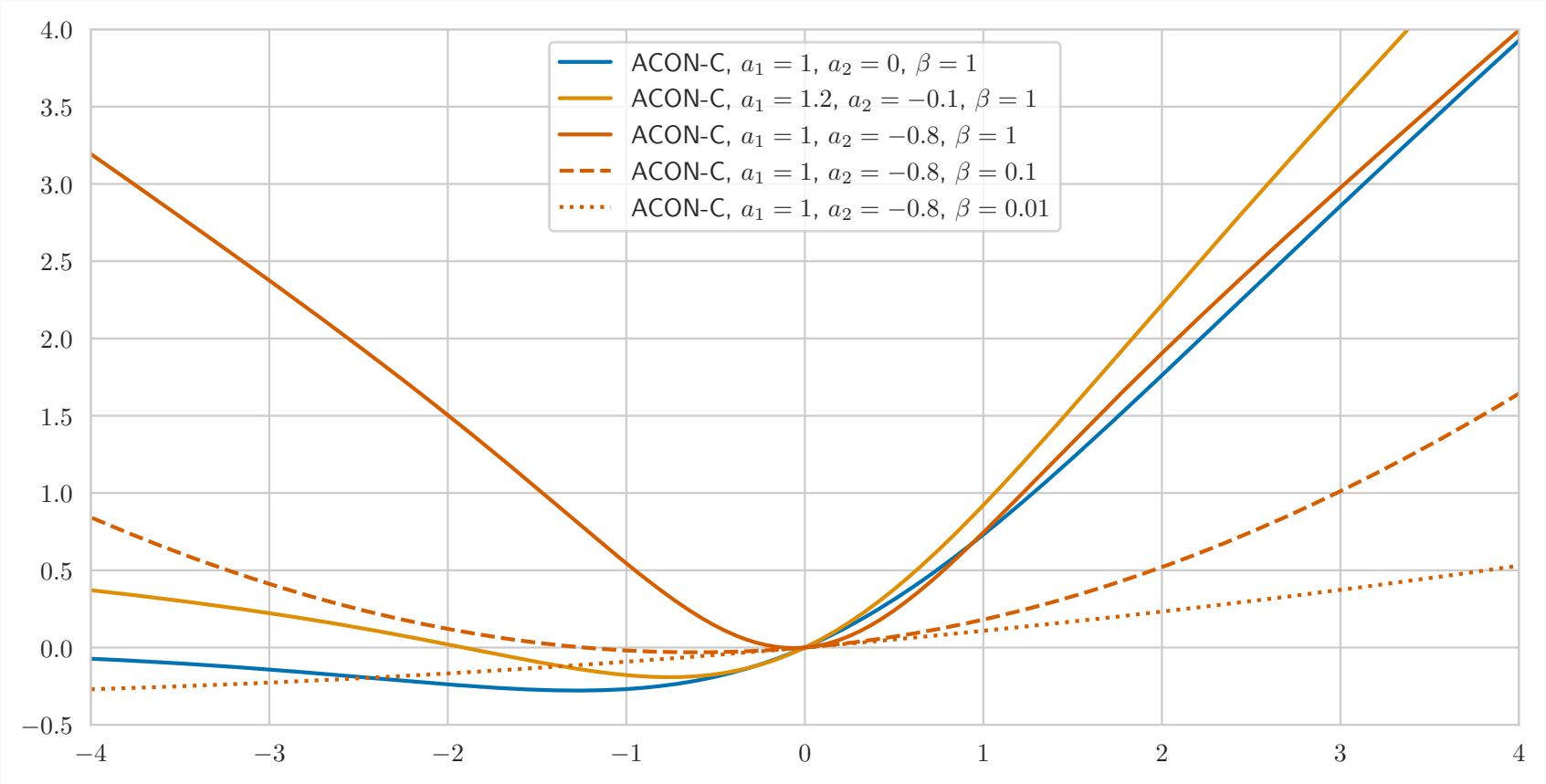
$$f_{ACON-C} = S_\beta(a_1 x, a_2 x) = (a_1 - a_2)\, x\sigma\left(\beta(a_1 - a_2)x\right) + a_2 x;$$

теперь a_1 и a_2 могут стать обучаемыми параметрами, а интуитивно это просто пределы производной с двух сторон:

$$\lim_{x\to\infty} \frac{\mathrm{d}f_{ACON-C}}{\mathrm{d}x} = a_1, \qquad \lim_{x\to-\infty} \frac{\mathrm{d}f_{ACON-C}}{\mathrm{d}x} = a_2.$$

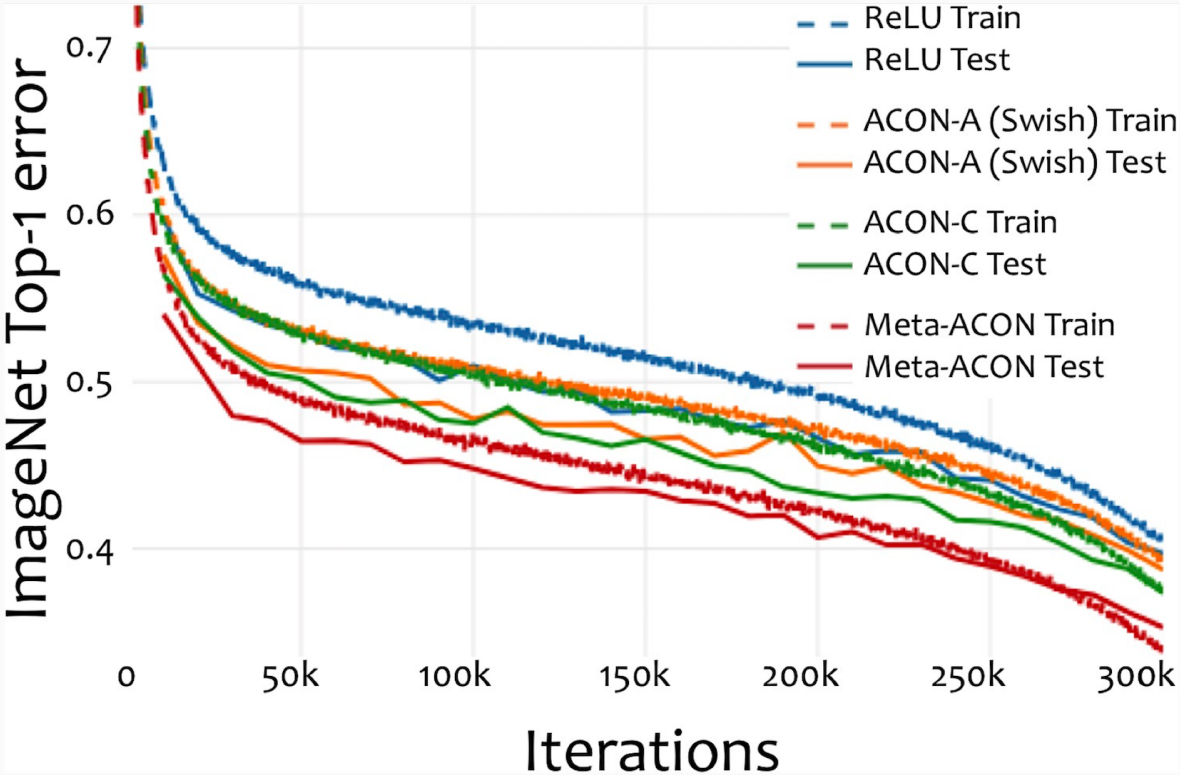
Функции активации

- Вот так выглядит ACON-C для разных значений параметров:



Функции активации

- Пишут, что подбором параметров можно очень много выгадать даже в самых стандартных избитых задачах:



- Но как-то с тех пор не произошло революции...
- Это частая история в DL.

Настоящие нейроны

- Есть ли градиентный спуск в настоящих нейронах?
- Скорее всего нет (увидим позже почему).
- Хотя у Хинтона есть интересный доклад об этом… но всё-таки вряд ли.
- *Обучение по Хеббу* – neurons that fire together, wire together:

$$\Delta w_i = \eta x_i x_j.$$

- В ML привело к *сетям Хопфилда*, которые вроде как реализуют ассоциативную память, но плохо.
- Новая тема: *spike-timing-dependent plasticity* (STDP), т.е. насколько спайки зависят от временных свойств сигналов.

От перцептрона к нейронной сети

Суть происходящего

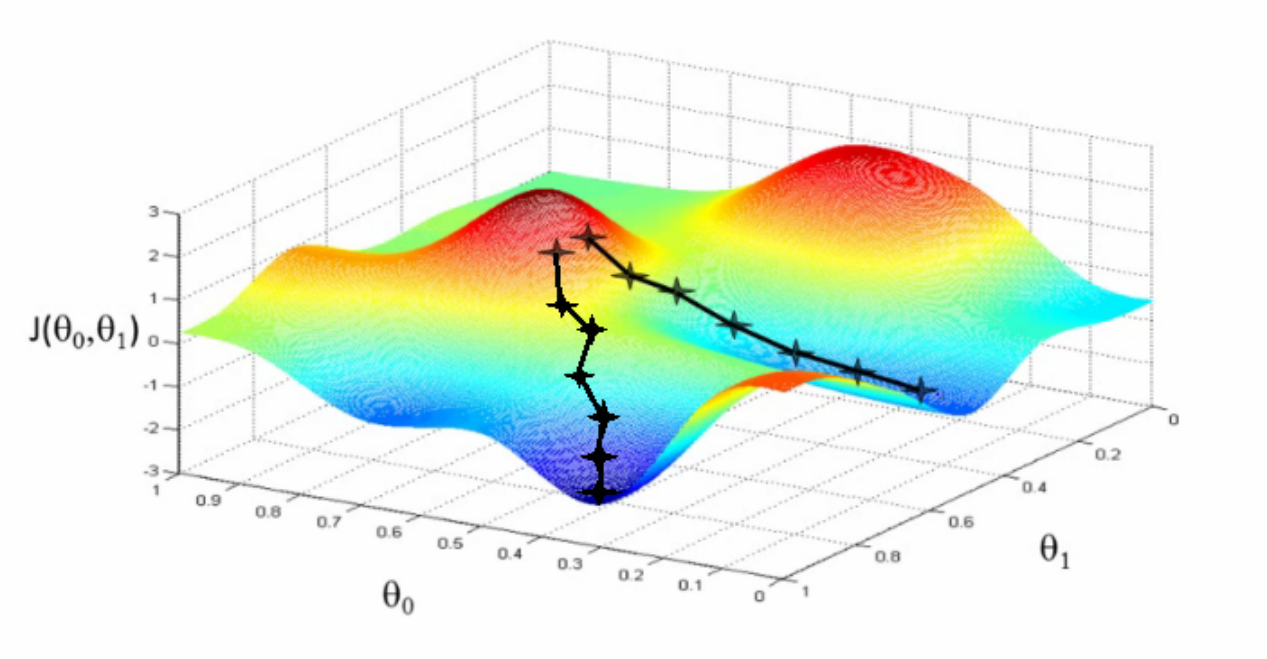
- Вообще говоря, всё, что мы делаем в машинном обучении, — это аппроксимация и оптимизация функций.
- Например:
  - есть функция из картинок в то, что на них изображено;
  - ещё проще: бинарная функция из пикселей картинки в «котик или не котик»;
  - функция, мягко скажем, довольно сложная; задана в виде датасета; как нам её реализовать?
- Обычный подход в машинном обучении:
  - строим какой-то класс моделей, обычно параметрический;
  - и подгоняем параметры так, чтобы модель хорошо описывала данные;
  - то есть оптимизируем какую-то функцию ошибки или функцию качества (правдоподобие, апостериорную вероятность).
- Нейронные сети — это очень мощный и гибкий класс таких моделей.
- Сложность в том, как же обучить их параметры.

Машинное обучение как оптимизация

- Итак, в машинном обучении много разных задач, с учителем и без.
- Их обычно решают по теореме Байеса, пересчитывая наши представления о параметрах в зависимости от полученных данных.
- Для этого обычно надо оптимизировать какую-нибудь сложную функцию (например, апостериорное распределение).
- И для невыпуклых функций это обычно делают тем или иным вариантом *градиентного спуска*.

Градиентный спуск

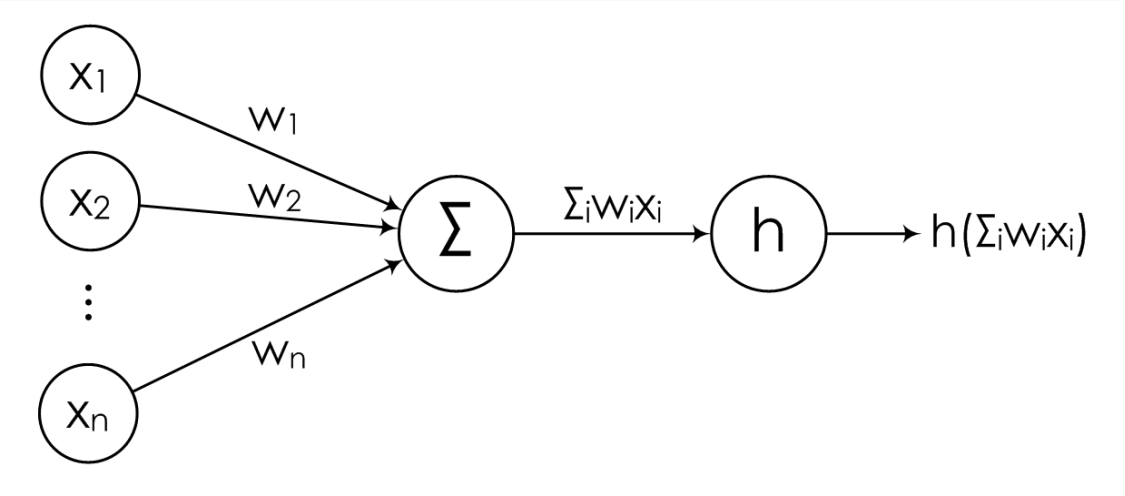
- Градиентный спуск — главный и фактически единственный способ оптимизации очень сложных функций.



- Берём градиент $\nabla E(\mathbf{w})$, сдвигаемся немножко, повторяем.

Перцептрон

- Основной компонент нейронной сети – *перцептрон*:



- Линейная комбинация входов, потом нелинейность:

$$y = h(\mathbf{w}^\top \mathbf{x}) = h\left(\sum_i w_i x_i\right).$$

Перцептрон

- Обучение *градиентным спуском*.
- Для исходного перцептрона Розенблатта ($h = \mathrm{id}$):

$$E_P(\mathbf{w}) = -\sum_{\mathbf{x}\in\mathcal{M}} y(\mathbf{x})\left(\mathbf{w}^\top \mathbf{x}\right),$$

$$\mathbf{w}^{(\tau+1)} = \mathbf{w}^{(\tau)} - \eta\nabla_{\mathbf{w}} E_P(\mathbf{w}) = \mathbf{w}^{(\tau)} + \eta t_n \mathbf{x}_n.$$

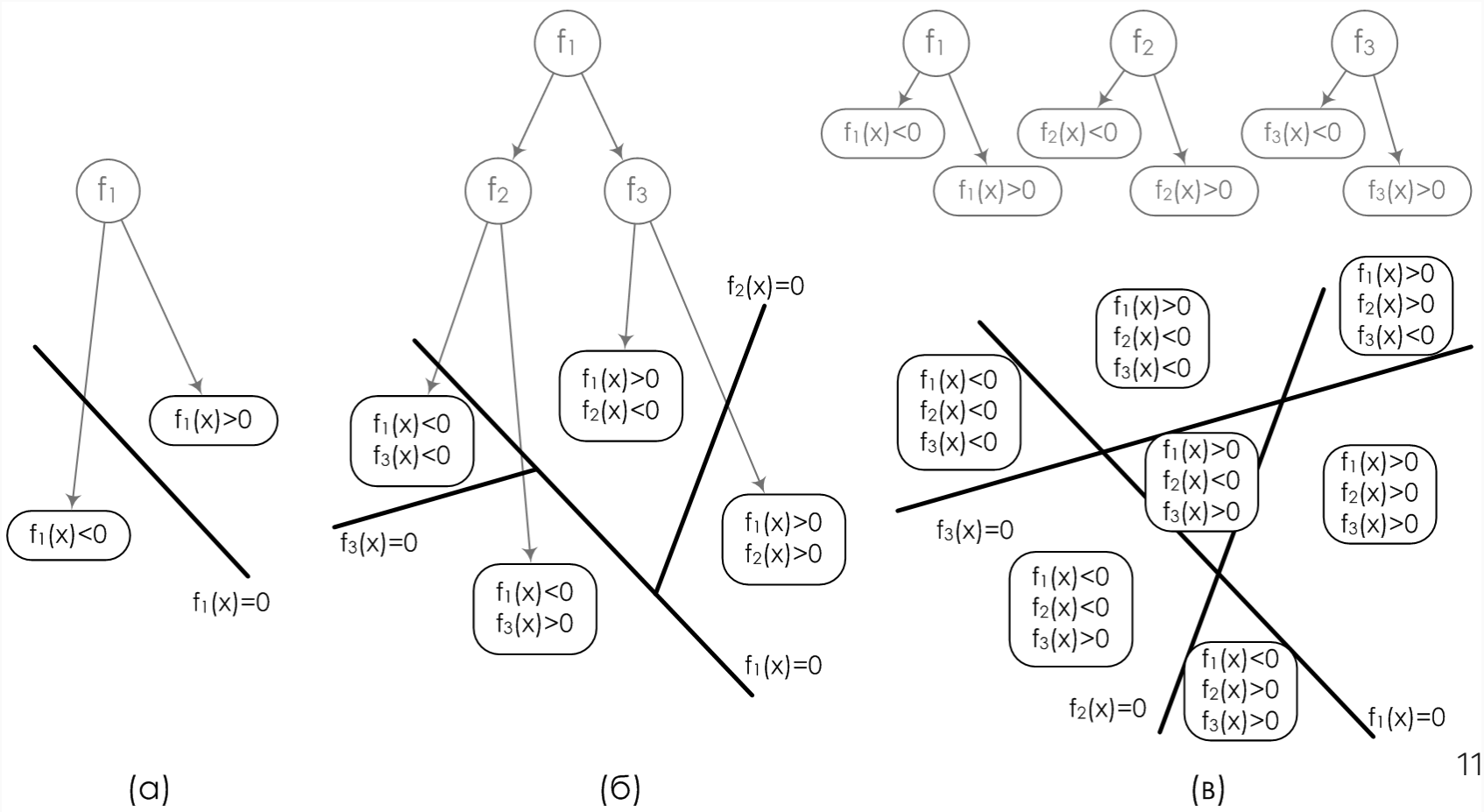
- Или, к примеру, можно делать классификацию, подставляя в качестве функции активации логистический сигмоид $h(x) = \sigma(x) = \frac{1}{1+e^{-x}}$:

$$E(\mathbf{w}) = -\frac{1}{N}\sum_{i=1}^{N}\left(y_i \log\sigma(\mathbf{w}^\top \mathbf{x}_i) + (1-y_i)\log\left(1-\sigma(\mathbf{w}^\top \mathbf{x}_i)\right)\right).$$

- По сути это просто логистическая регрессия.

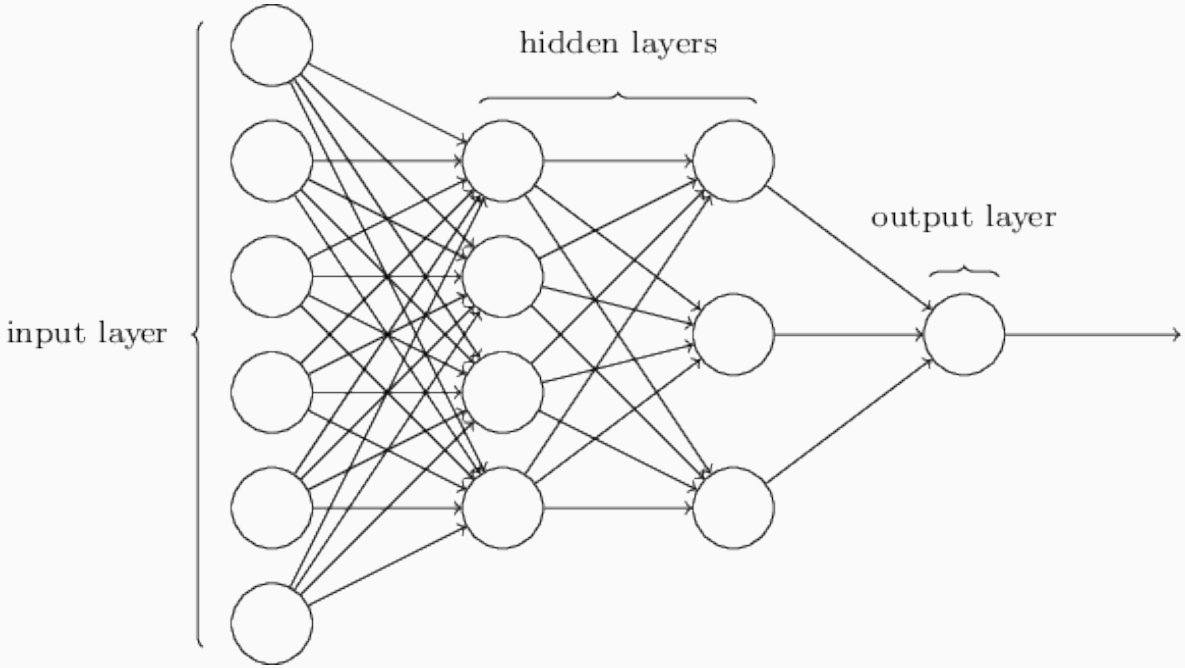
Объединяем перцептроны в сеть

- Сеть перцептронов; выходы одних – входы других.
- Hornik, 1990: двух уровней достаточно для приближения любой функции.
- Но глубокие сети эффективнее — distributed representations:



Объединяем перцептроны в сеть

- Обычно нейронные сети организованы в *слои*.
- Это очень естественно и легко параллелизуется.



Объединяем перцептроны в сеть

- Но по сути мы приближаем очень сложную функцию большой композицией простых функций.
- Как теперь её обучить?
- Ответ прост: градиентный спуск. Но есть и сложности.

Градиентный спуск и графы вычислений

Градиентный спуск

- 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 пройтись, чтобы хоть куда-то сдвинуться?..

Градиентный спуск

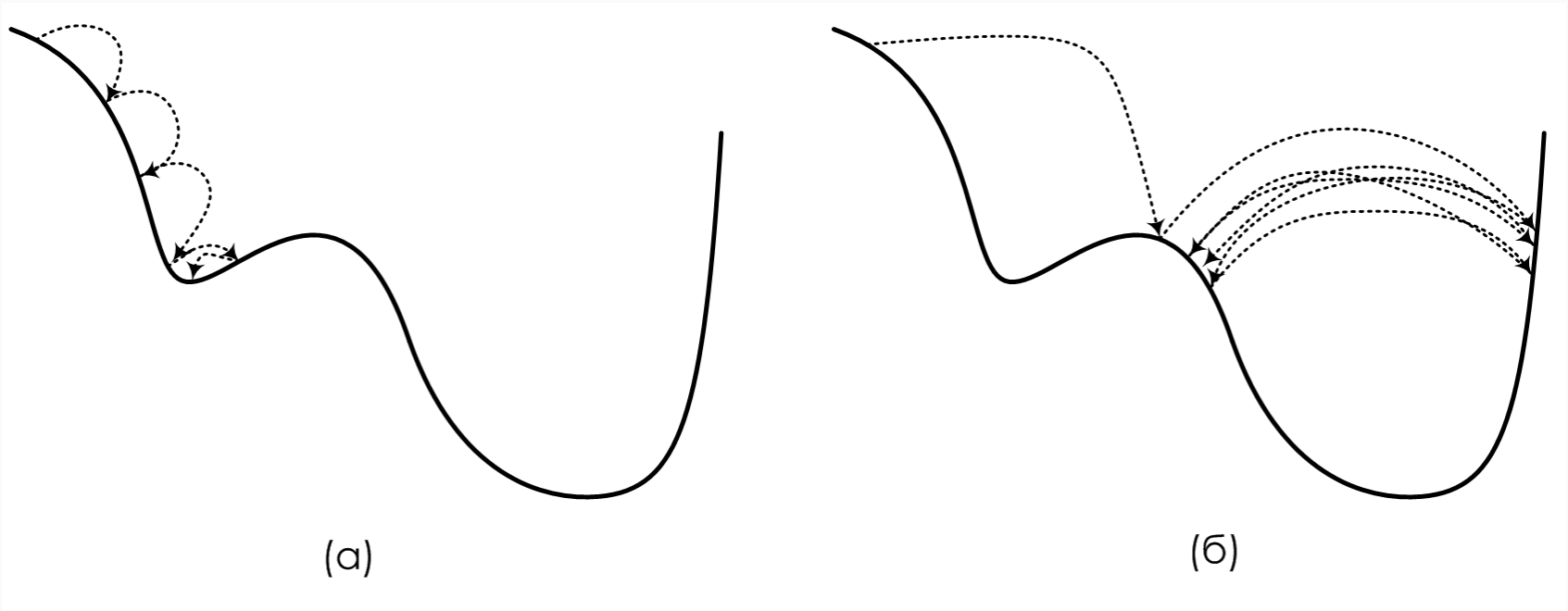
- Нет, конечно — *стохастический градиентный спуск* обновляет после каждого примера:

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

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

Градиентный спуск

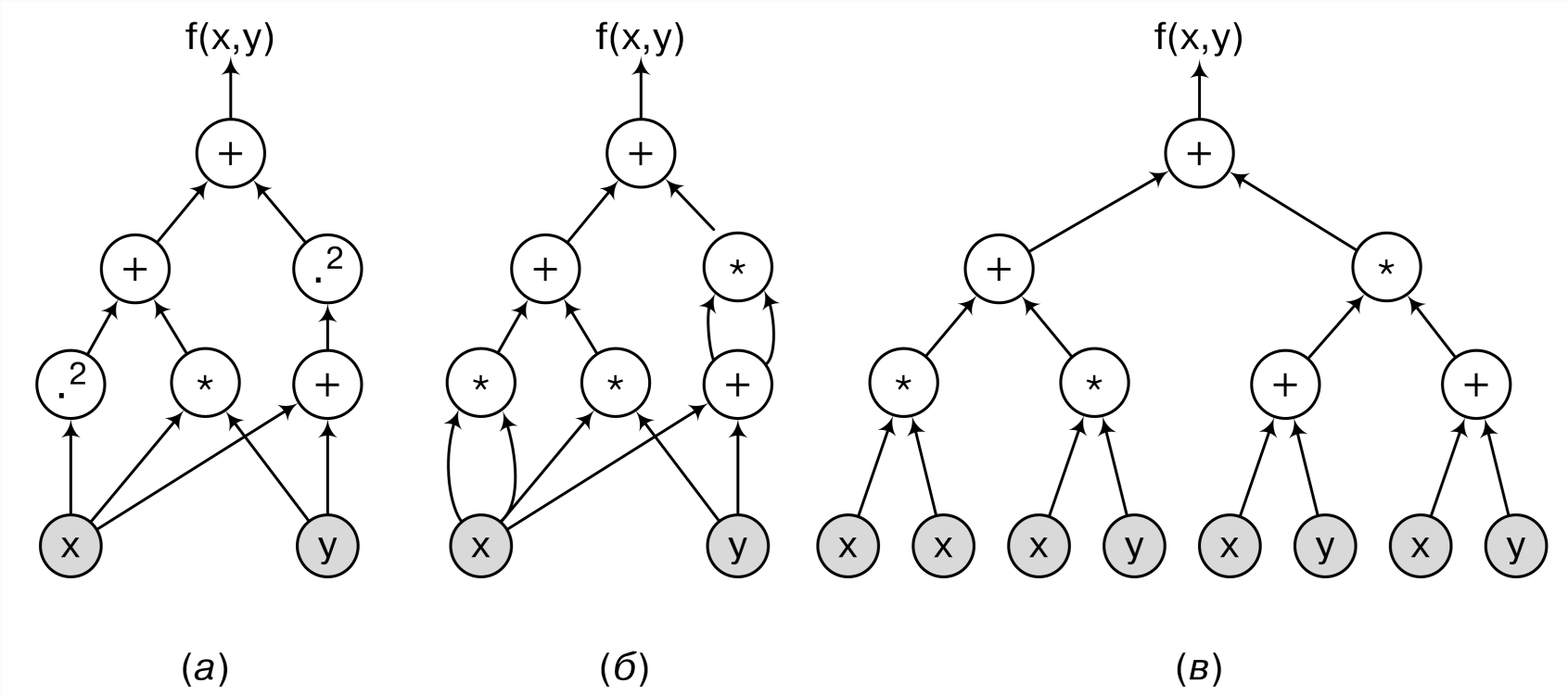
- Со скоростью обучения η масса проблем:



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

Граф вычислений, frop and bprop

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



Граф вычислений, fprop and bprop

- 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:

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

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

Граф вычислений, frop and bprop

- 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:

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

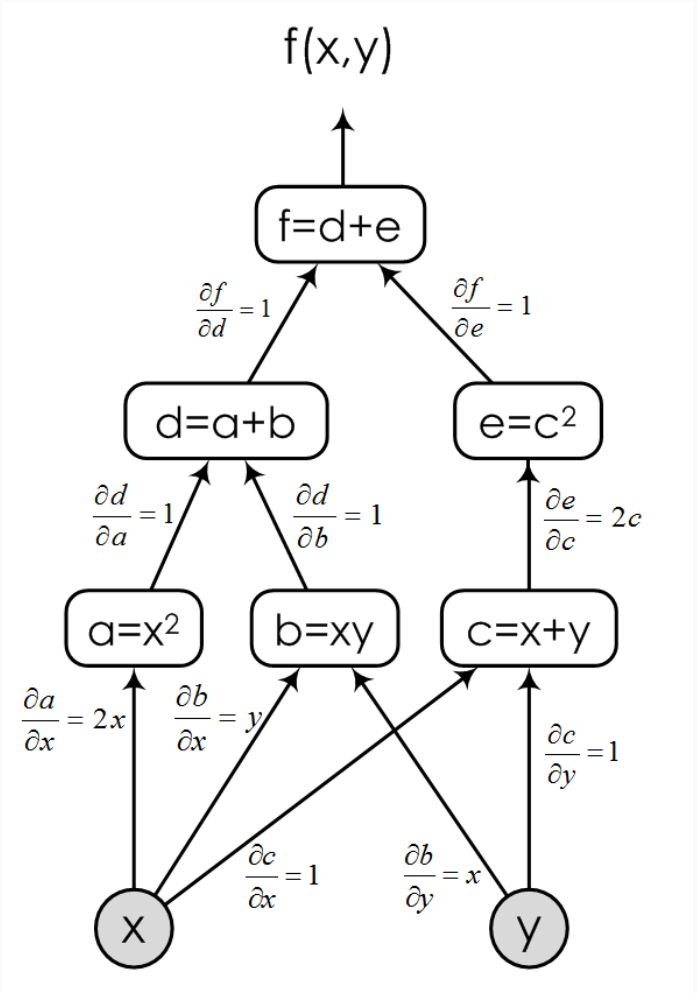
$$\nabla_{\mathbf{x}} f = \frac{\partial f}{\partial g_1}\nabla_{\mathbf{x}} g_1 + \dots + \frac{\partial f}{\partial g_k}\nabla_{\mathbf{x}} g_k = \sum_{i=1}^{k}\frac{\partial f}{\partial g_i}\nabla_{\mathbf{x}} g_i.$$

- Это матричное умножение на *матрицу Якоби*:

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

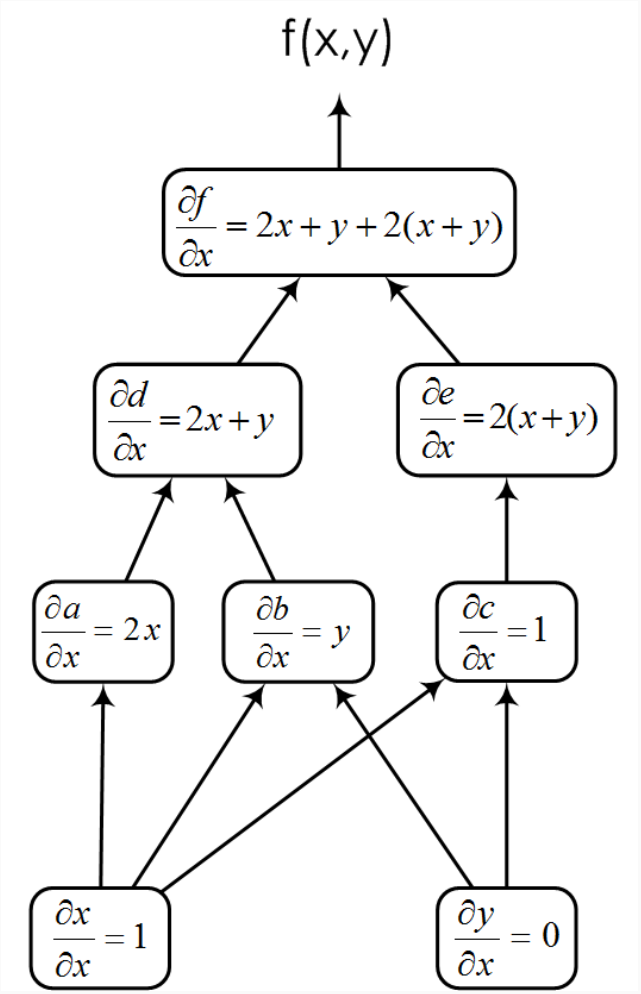
Граф вычислений, frop and bprop

- Возвращаемся к примеру:



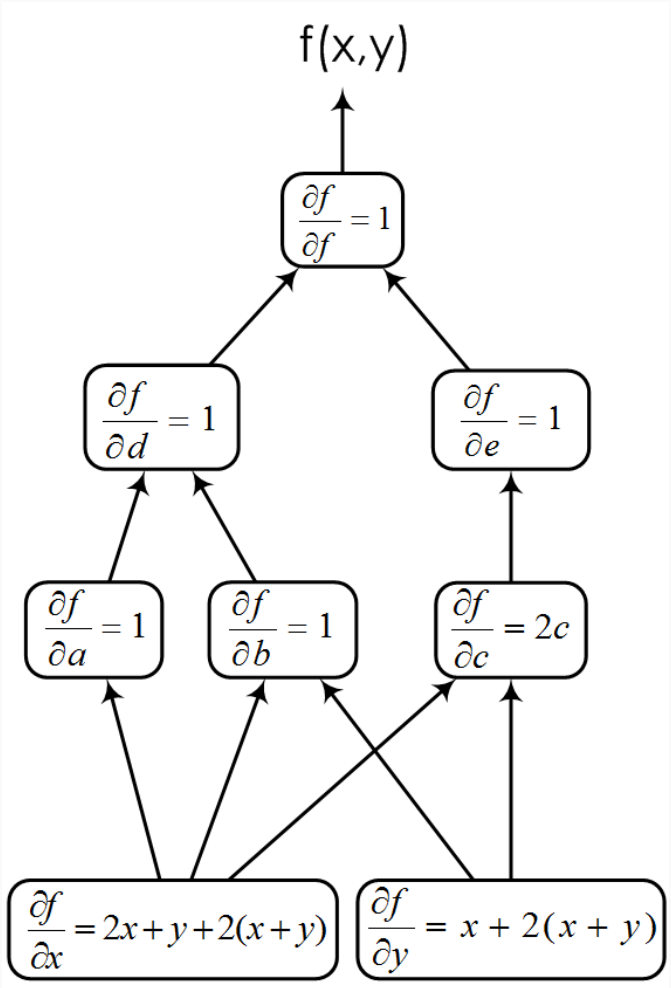
Граф вычислений, frop and bprop

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



Граф вычислений, frop and bprop

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

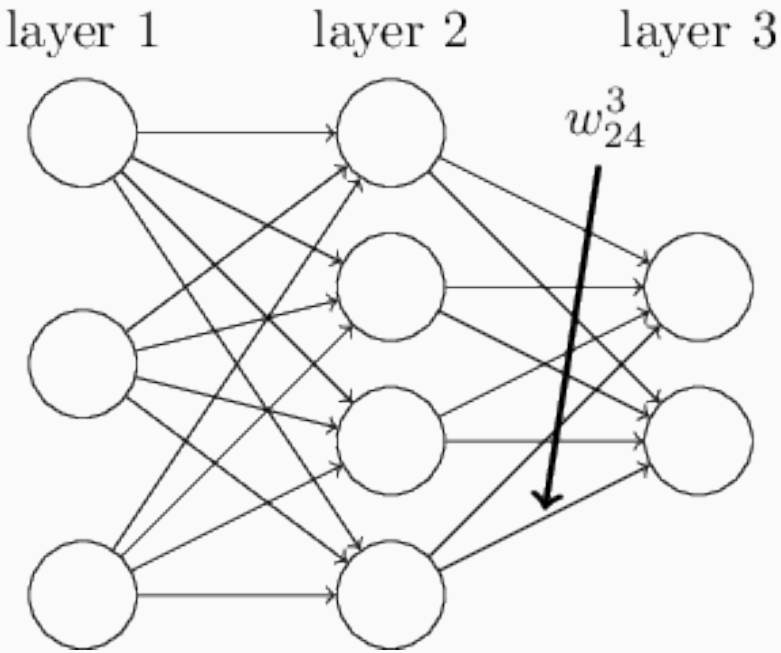


Граф вычислений, frop and bprop

- 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.

Граф вычислений, frop and bprop

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



w^l_{jk} is the weight from the k^{th} neuron in the $(l-1)^{\text{th}}$ layer to the j^{th} neuron in the l^{th} layer

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

Граф вычислений, frop and bprop

- 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: $\frac{\partial C}{\partial w_{jk}^{(l)}}$, $\frac{\partial C}{\partial b_j^{(l)}}$.

Граф вычислений, frop and bprop

- 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.

Граф вычислений, fprop and bprop

- 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

Градиентный спуск

- «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):

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

Градиентный спуск

- 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):

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

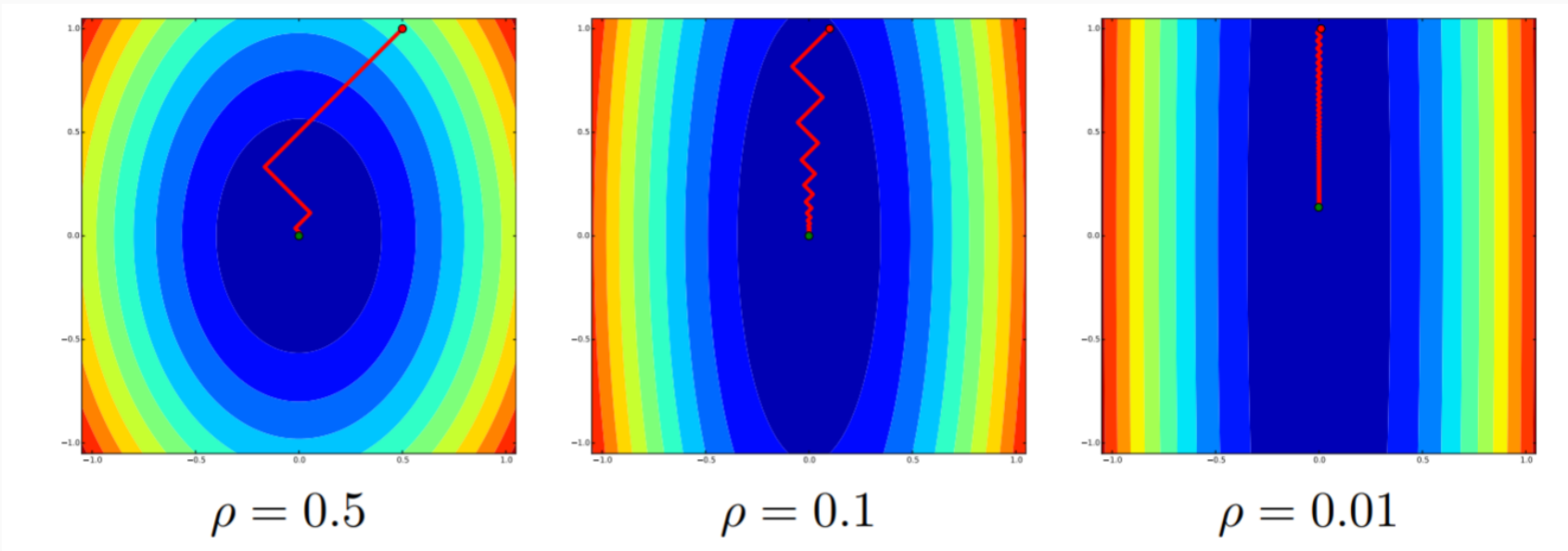
  - или даже более сильным условиям Вульфа (Wolfe rule): Армихо плюс

$$|\phi_k'(\alpha)| \leq c_2 |\phi_k'(0)|,$$

    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 $f(x, y) = \frac{1}{2}x^2 + \frac{\rho}{2}y^2 \to \min_{x,y}$



$\rho = 0.5$ $\rho = 0.1$ $\rho = 0.01$

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

Градиентный спуск

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

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

- Здесь тоже применимо условие Армихо:

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

- Было бы круто! Но H_k посчитать просто нереально.

Градиентный спуск

- 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*:

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

  - минимизация эмпирического риска

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

  - минимизация вариационной нижней оценки (ELBO)... но об этом позже.

- Что такое теперь, получается, мини-батчи?

Стохастический градиентный спуск

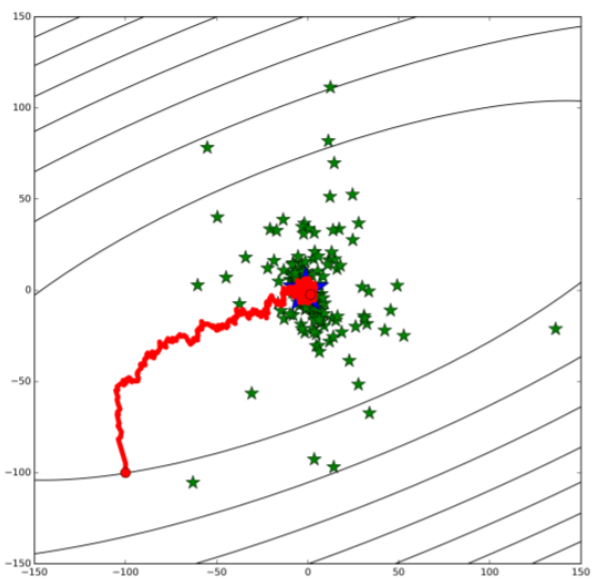
- Да просто эмпирические оценки общей функции по подвыборке:

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

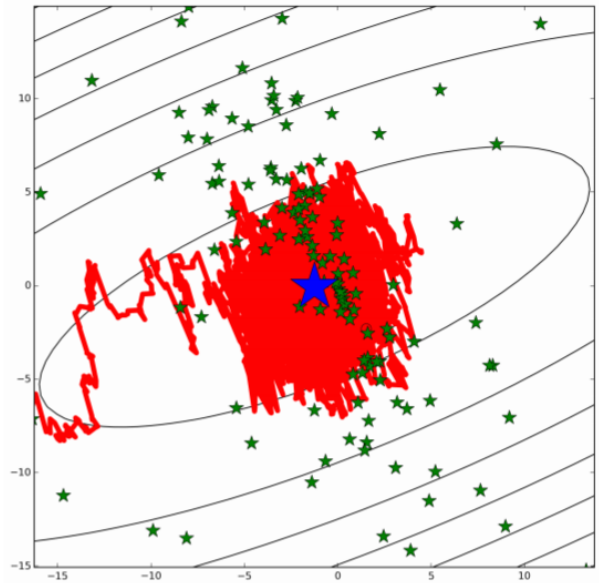
- Это очень хорошие оценки: несмещённые, сходятся на бесконечности (правда, медленно), легко посчитать.
- В целом, так и мотивируется стохастический градиентный спуск (SGD): метод Монте-Карло по сути.
- Но есть проблемы...

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

Стохастический градиентный спуск

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

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

- Давайте оценим невязку точки на очередной итерации:

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

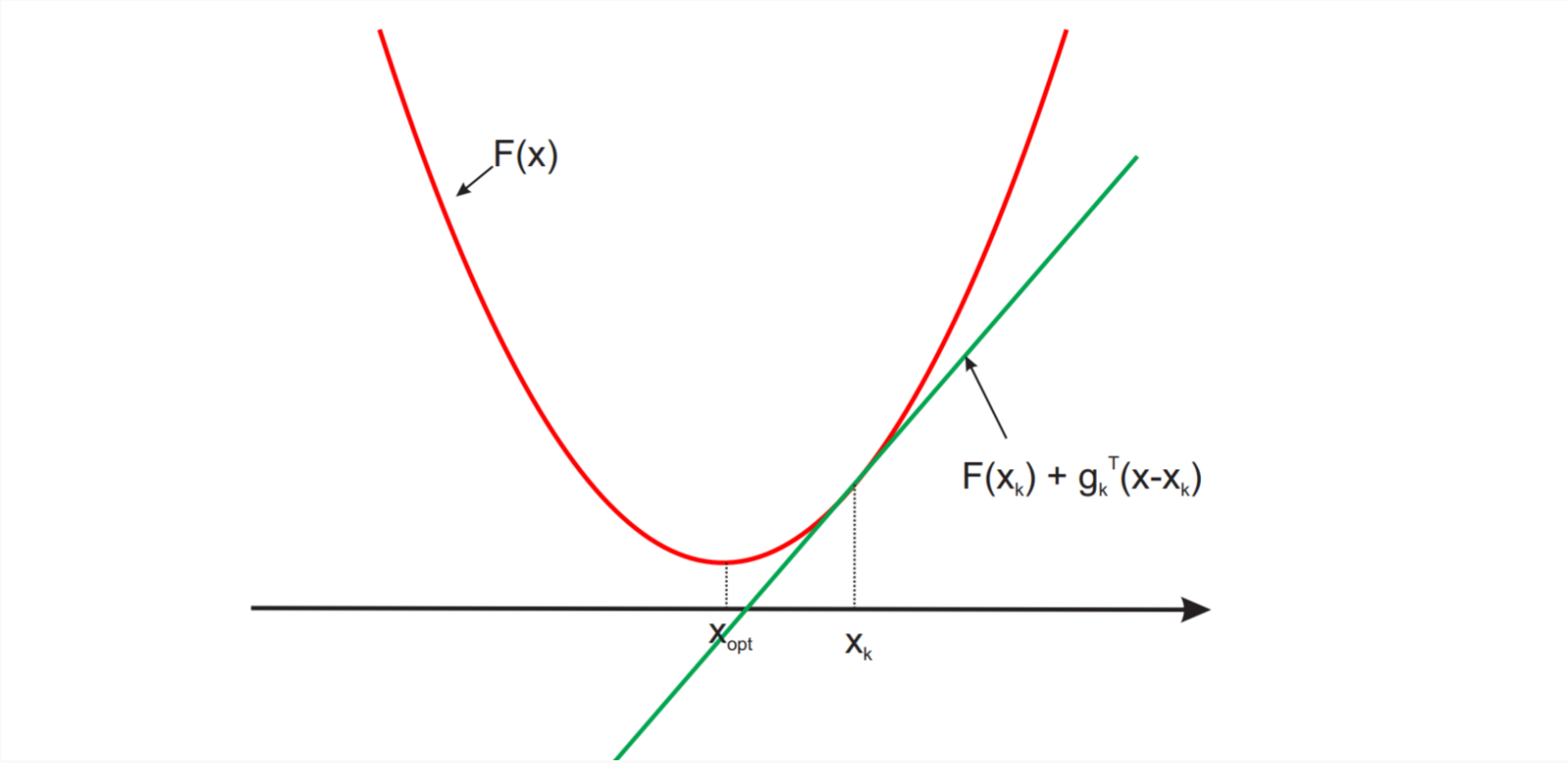
- Возьмём ожидание по $q(\mathbf{y})$ в момент времени k:

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

Стохастический градиентный спуск

- Для простоты предположим, что F выпуклая:

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



Стохастический градиентный спуск

- У нас было

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

- Значит,

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

Стохастический градиентный спуск

- Возьмём ожидание от левой части и просуммируем:

$$\sum_{i=0}^{k} \alpha_i (\mathbb{E}F(\mathbf{x}_i) - F(\mathbf{x}_{\text{opt}})) \leq$$

$$\leq \frac{1}{2}\|\mathbf{x}_0 - \mathbf{x}_{\text{opt}}\|^2 + \frac{1}{2}\sum_{i=0}^{k} \alpha_i^2 \mathbb{E}\|\hat{\mathbf{g}}_i\|^2 - \frac{1}{2}\mathbb{E}\|\mathbf{x}_{k+1} - \mathbf{x}_{\text{opt}}\|^2 \leq$$

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

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

Стохастический градиентный спуск

- Воспользуемся выпуклостью:

$$\mathbb{E}F\left(\frac{\sum_i \alpha_i \mathbf{x}_i}{\sum_i \alpha_i}\right) - F(\mathbf{x}_{\text{opt}}) \leq$$

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

- 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

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

Стохастический градиентный спуск

- 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 $\alpha_i = h$

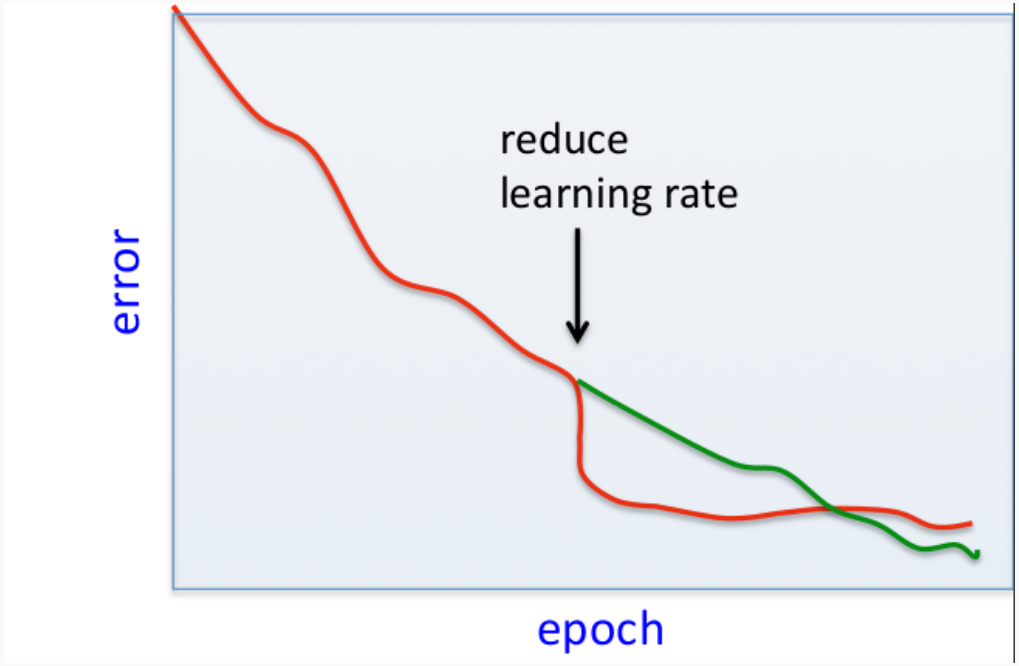
$$\mathbb{E}F(\hat{\mathbf{x}}_k) - F(\mathbf{x}_{\text{opt}}) \leq \frac{R^2}{2h(k+1)} + \frac{G^2 h}{2} \to_{k\to\infty} \frac{G^2 h}{2}.$$

Стохастический градиентный спуск

- 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.

Метод моментов

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



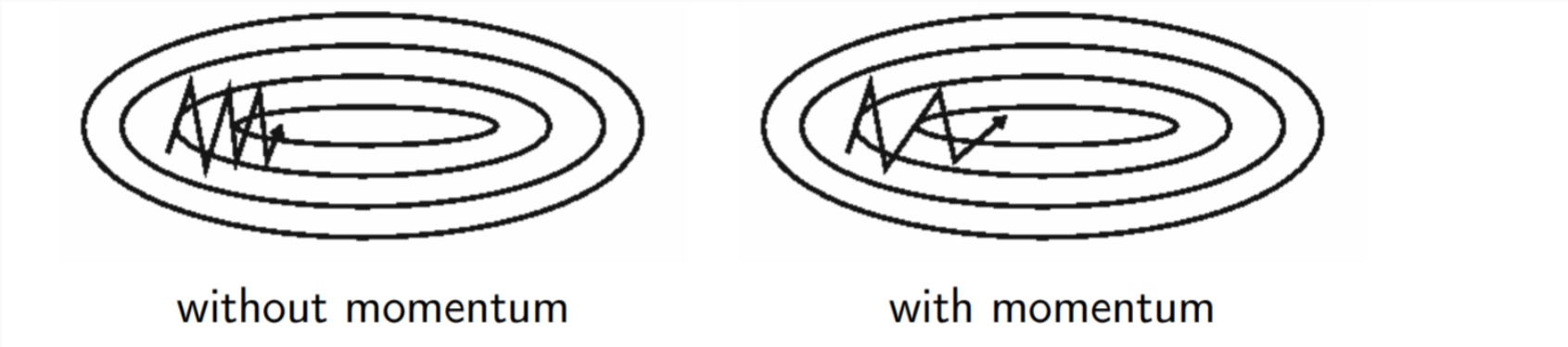
- Но это в любом случае никак не учитывает собственно F.

Метод моментов

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

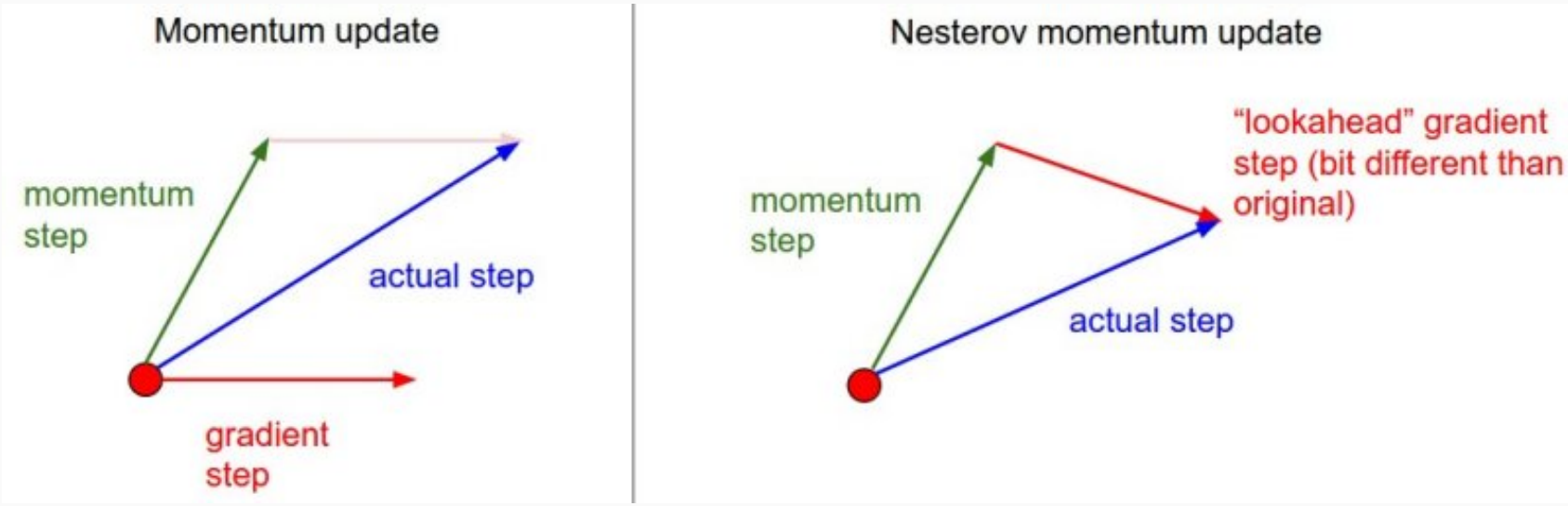
$$u_t = \gamma u_{t-1} + \eta \nabla_{\mathbf{x}} F(\mathbf{x}),$$
$$\mathbf{x} = \mathbf{x} - u_t.$$

- И теперь мы сохраняем γu_{t-1}.



Метод моментов

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



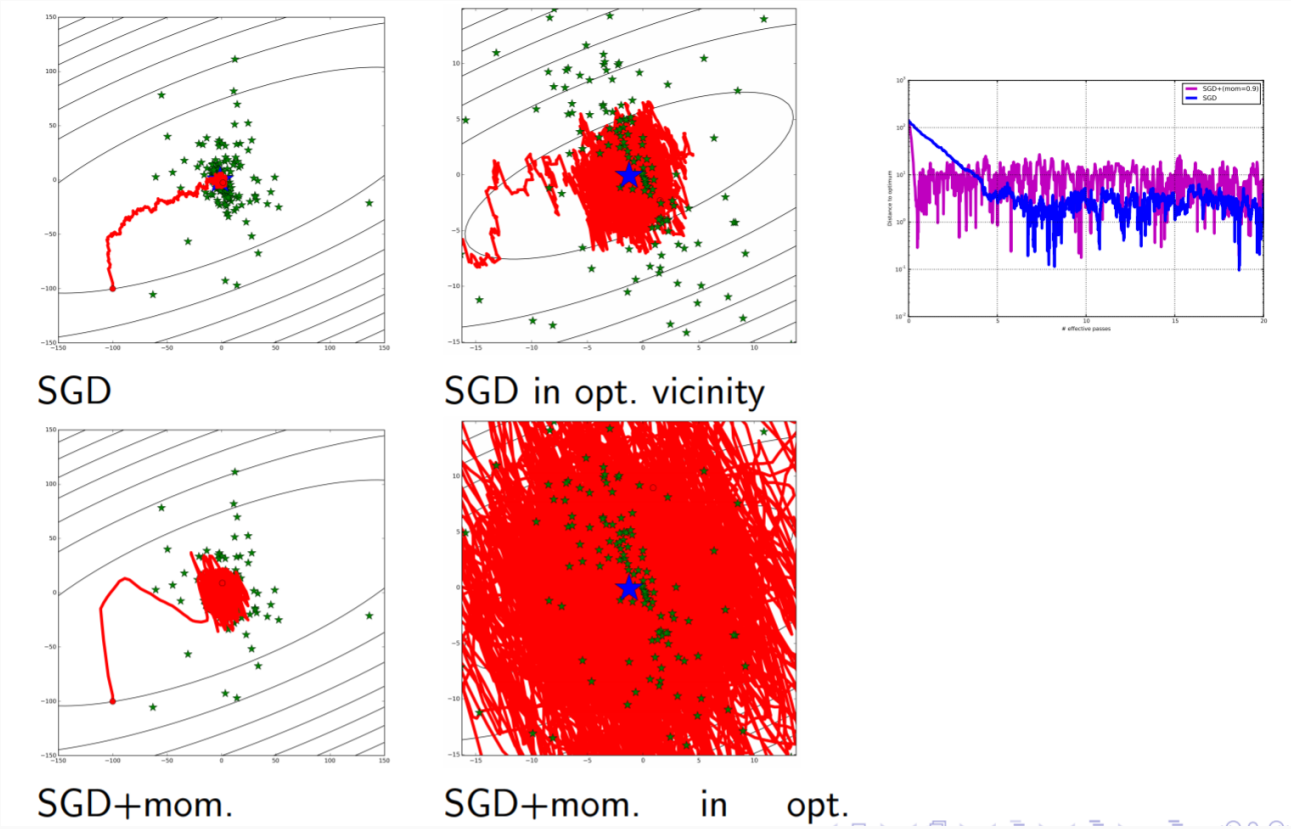
Метод моментов

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

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

Метод моментов

- Всё равно, конечно, проблемы не пропадают:



- Можно ли ещё лучше?..

Адаптивные методы градиентного спуска

- 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.

Адаптивные методы градиентного спуска

- *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 $\mathbf{w}_i$.

Адаптивные методы градиентного спуска

- 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:

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

Адаптивные методы градиентного спуска

- 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.

Адаптивные методы градиентного спуска

- 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:

$$\mathbb{E}\left[\Delta\mathbf{w}^2\right]_t = \rho\mathbb{E}\left[\Delta\mathbf{w}^2\right]_{t-1} + (1-\rho)\Delta\mathbf{w}^2, \text{ где}$$

$$u_t = -\frac{\sqrt{\mathbb{E}\left[\Delta\mathbf{w}^2\right]_{t-1} + \epsilon}}{\sqrt{G_{t-1} + \epsilon}} \cdot g_{t-1}.$$

Адаптивные методы градиентного спуска

- 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:

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

Адаптивные методы градиентного спуска

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

$$
\begin{aligned}
m_t &= \beta_1 m_{t-1} + (1 - \beta_1) g_t, \\
v_t &= \beta_2 v_{t-1} + (1 - \beta_2) g_t^2, \\
u_t &= \frac{\eta}{\sqrt{v + \epsilon}} m_t.
\end{aligned}
$$

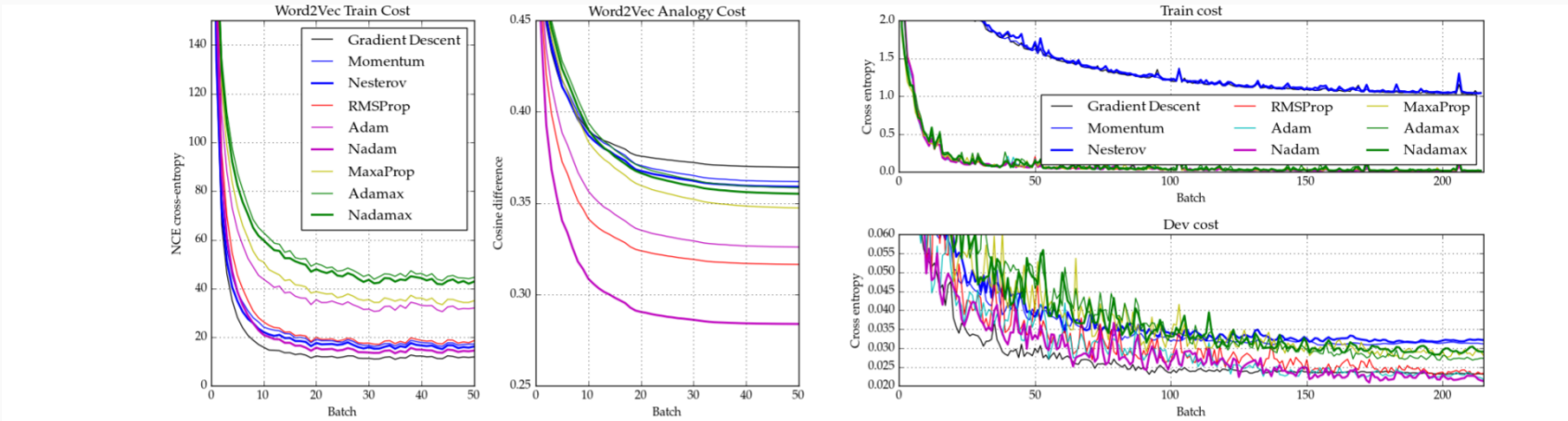
- (Kingma, Ba, 2014) рекомендуют $\beta_1 = 0.9$, $\beta_2 = 0.999$, $\epsilon = 10^{-8}$.
- *Adam* практически не требует настройки, используется на практике очень часто.
- ...[`http://ruder.io/optimizing-gradient-descent/`]...

Адаптивные методы градиентного спуска

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

Algorithm 8 Nesterov-accelerated adaptive moment estimation

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



Адаптивные методы градиентного спуска

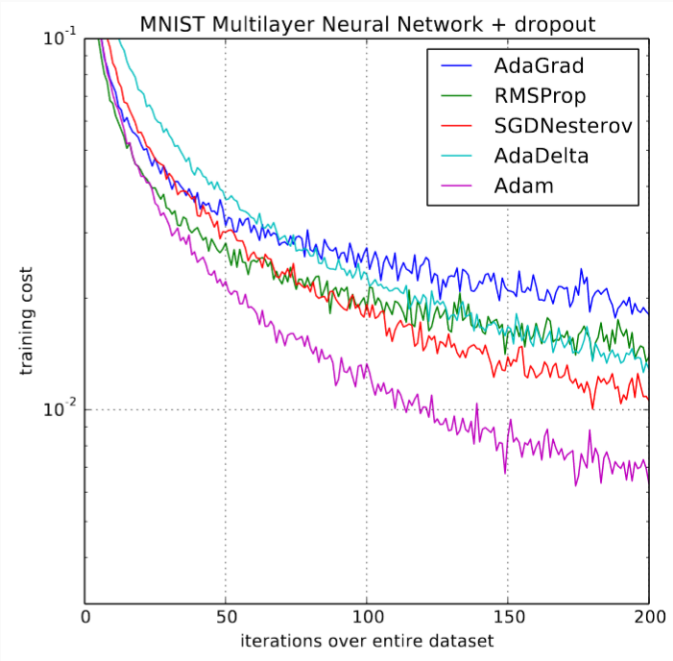
- Чуть более общий взгляд – всё это выглядит вот так:
  - обычный стохастический градиентный спуск:
    $\mathbf{w}_{k+1} = \mathbf{w}_k - \alpha_k \tilde{\nabla} f(\mathbf{w}_k)$, где $\tilde{\nabla} f(\mathbf{w}_k) = \nabla f(\mathbf{w}_k; \mathbf{x}_{i_k})$;
  - SGD с моментами:
    $\mathbf{w}_{k+1} = \mathbf{w}_k - \alpha_k \tilde{\nabla} f\left(\mathbf{w}_k + \gamma_k \left(\mathbf{w}_k - \mathbf{w}_{k-1}\right)\right) + \beta_k \left(\mathbf{w}_k - \mathbf{w}_{k-1}\right)$;
  - адаптивный SGD:

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

    где обычно $H_k = \operatorname{diag}\left(\left[\sum_{i=1}^{k} \eta_i \mathbf{g}_i \circ \mathbf{g}_i\right]^{1/2}\right)$, где $\mathbf{g}_k = \tilde{\nabla} f\left(\mathbf{w}_k + \gamma_k \left(\mathbf{w}_k - \mathbf{w}_{k-1}\right)\right)$ (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 $H_k = \mathbf{I}$.

Adam, AdamW и другие животные

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



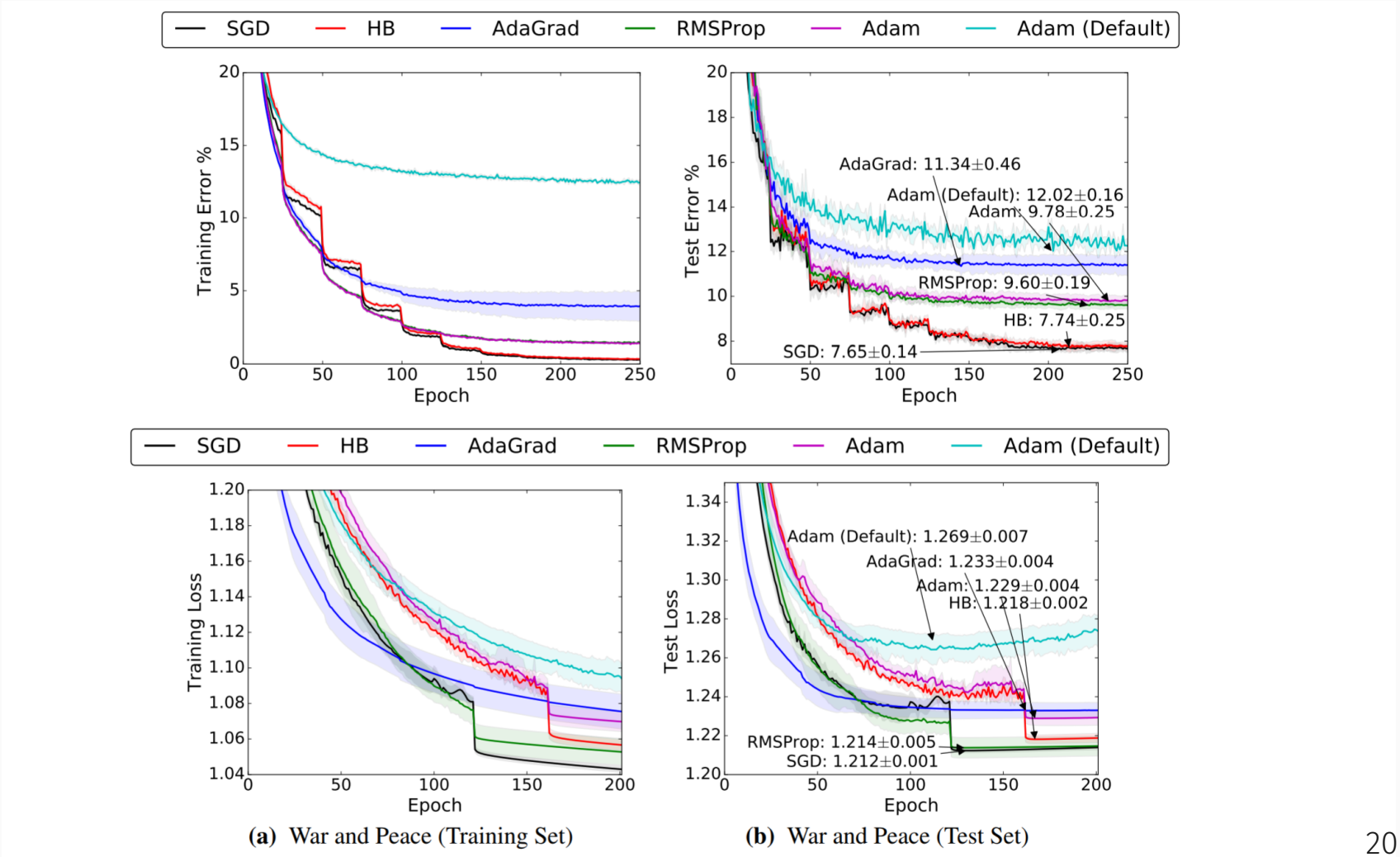
- Но потом выяснилось, что не всё так просто... часто применяли простой SGD. Почему?

Adam, AdamW и другие животные

- 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

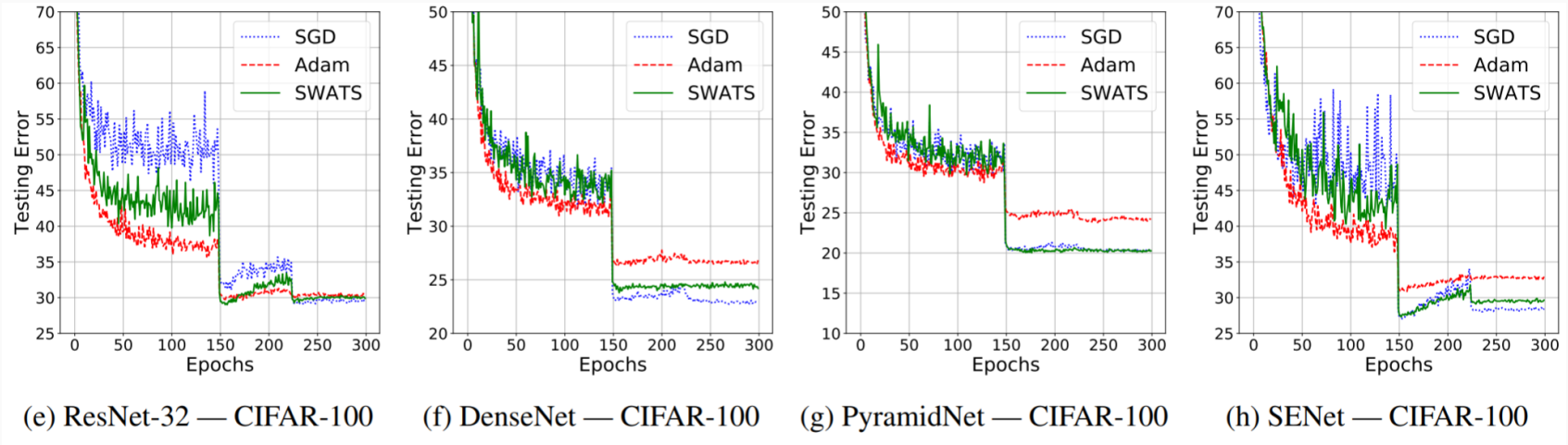
- The Marginal Value of Adaptive Gradient Methods in Machine Learning (Wilson et al., May 2017)



(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...

Adam, AdamW и другие животные

- 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.

Adam, AdamW и другие животные

- 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...

Adam, AdamW и другие животные

- ...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 $\boldsymbol{x}_{t=0} \in \mathbb{R}^n$, first moment vector $\boldsymbol{m}_{t=0} \leftarrow \boldsymbol{0}$, schedule multiplier $\eta_{t=0} \in \mathbb{R}$
3: **repeat**
4: $t \leftarrow t + 1$
5: $\nabla f_t(\boldsymbol{x}_{t-1}) \leftarrow \text{SelectBatch}(\boldsymbol{x}_{t-1})$ ▷ select batch and return the corresponding gradient
6: $\boldsymbol{g}_t \leftarrow \nabla f_t(\boldsymbol{x}_{t-1}) + w\boldsymbol{x}_{t-1}$
7: $\eta_t \leftarrow \text{SetScheduleMultiplier}(t)$ ▷ can be fixed, decay, be used for warm restarts
8: $\boldsymbol{m}_t \leftarrow \beta_1 \boldsymbol{m}_{t-1} + \eta_t \alpha \boldsymbol{g}_t$
9: $\boldsymbol{x}_t \leftarrow \boldsymbol{x}_{t-1} - \boldsymbol{m}_t - \eta_t w \boldsymbol{x}_{t-1}$
10: **until** *stopping criterion is met*
11: **return** optimized parameters $\boldsymbol{x}_t$

- 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.

Adam, AdamW и другие животные

- То же самое происходит и с Adam:

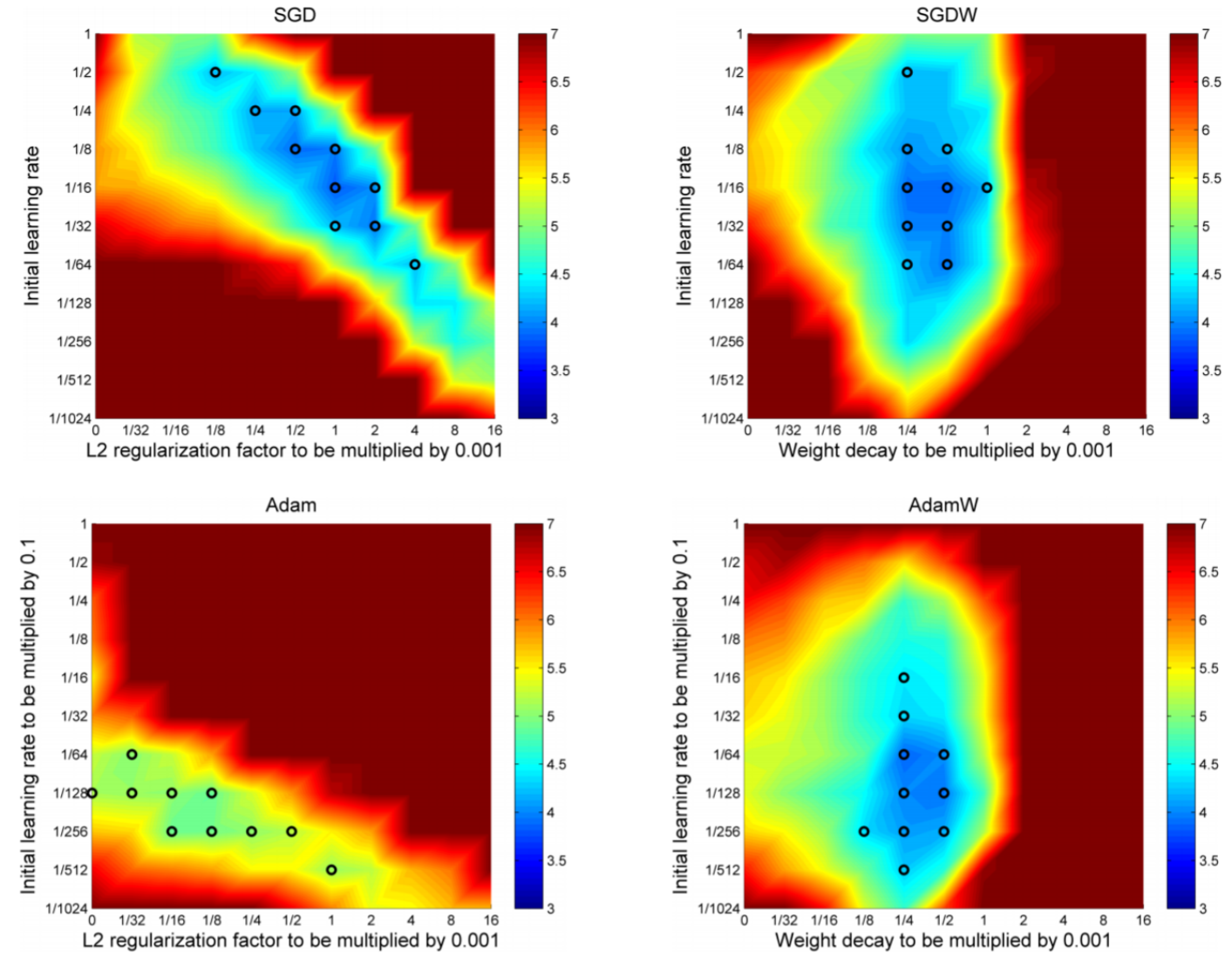
Algorithm 2 Adam with L_2 regularization and Adam with weight decay (AdamW)

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 $\boldsymbol{x}_{t=0} \in \mathbb{R}^n$, first moment vector $\boldsymbol{m}_{t=0} \leftarrow \boldsymbol{0}$, second moment vector $\boldsymbol{v}_{t=0} \leftarrow \boldsymbol{0}$, schedule multiplier $\eta_{t=0} \in \mathbb{R}$
3: **repeat**
4: $t \leftarrow t + 1$
5: $\nabla f_t(\boldsymbol{x}_{t-1}) \leftarrow \text{SelectBatch}(\boldsymbol{x}_{t-1})$ ▷ select batch and return the corresponding gradient
6: $\boldsymbol{g}_t \leftarrow \nabla f_t(\boldsymbol{x}_{t-1}) + w\boldsymbol{x}_{t-1}$
7: $\boldsymbol{m}_t \leftarrow \beta_1 \boldsymbol{m}_{t-1} + (1 - \beta_1)\boldsymbol{g}_t$ ▷ here and below all operations are element-wise
8: $\boldsymbol{v}_t \leftarrow \beta_2 \boldsymbol{v}_{t-1} + (1 - \beta_2)\boldsymbol{g}_t^2$
9: $\hat{\boldsymbol{m}}_t \leftarrow \boldsymbol{m}_t / (1 - \beta_1^t)$ ▷ β_1 is taken to the power of t
10: $\hat{\boldsymbol{v}}_t \leftarrow \boldsymbol{v}_t / (1 - \beta_2^t)$ ▷ β_2 is taken to the power of t
11: $\eta_t \leftarrow \text{SetScheduleMultiplier}(t)$ ▷ can be fixed, decay, or also be used for warm restarts
12: $\boldsymbol{x}_t \leftarrow \boldsymbol{x}_{t-1} - \eta_t \left(\alpha \hat{\boldsymbol{m}}_t / (\sqrt{\hat{\boldsymbol{v}}_t} + \epsilon) + w\boldsymbol{x}_{t-1} \right)$
13: **until** *stopping criterion is met*
14: **return** optimized parameters $\boldsymbol{x}_t$

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

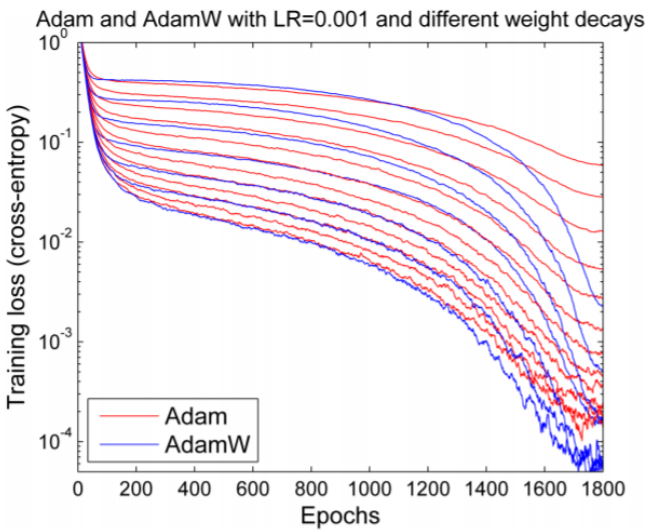
Adam, AdamW и другие животные

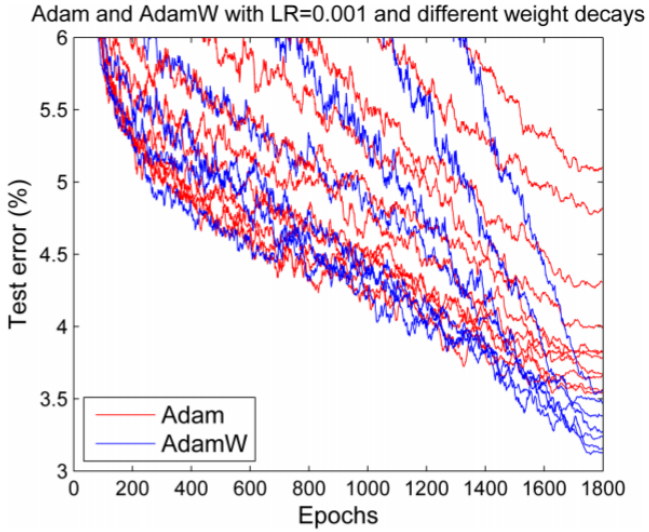
- И теперь гиперпараметры разделяются лучше:

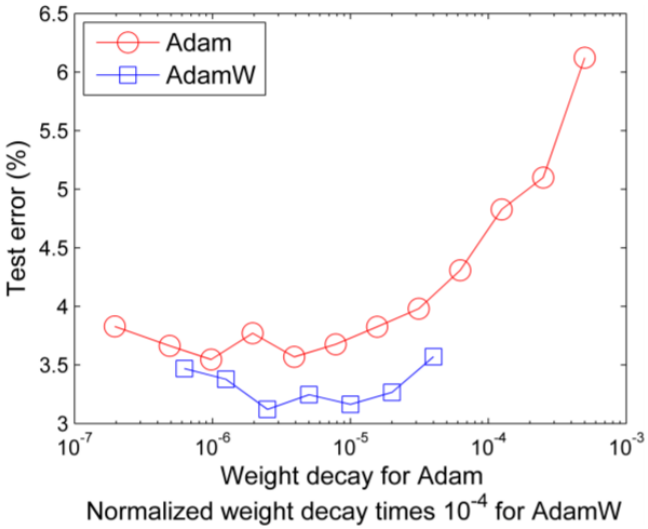


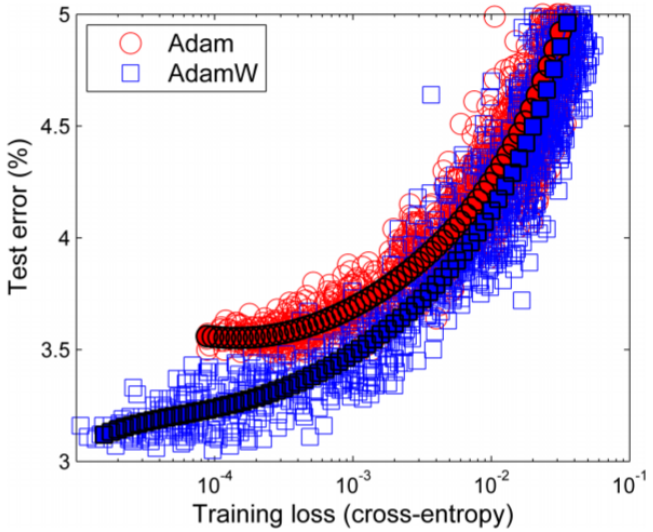
Adam, AdamW и другие животные

- Да и обобщается лучше:



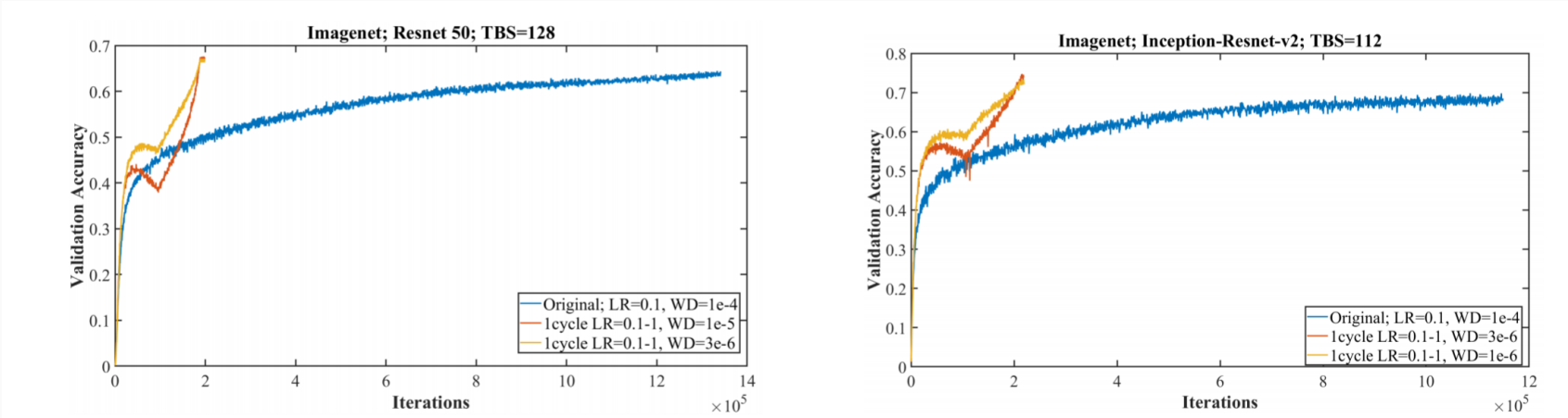






Super-Convergence и amsgrad

- 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.

Super-Convergence и amsgrad

- 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**
$g_t = \nabla f_t(x_t)$
$m_t = \beta_{1t} m_{t-1} + (1 - \beta_{1t}) g_t$
$v_t = \beta_2 v_{t-1} + (1 - \beta_2) g_t^2$
$\hat{v}_t = \max(\hat{v}_{t-1}, v_t)$ and $\hat{V}_t = \mathrm{diag}(\hat{v}_t)$
$x_{t+1} = \Pi_{\mathcal{F}, \sqrt{\hat{V}_t}}(x_t - \alpha_t m_t / \sqrt{\hat{v}_t})$
end for

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

Регуляризация в нейронных сетях

Регуляризация в нейронных сетях

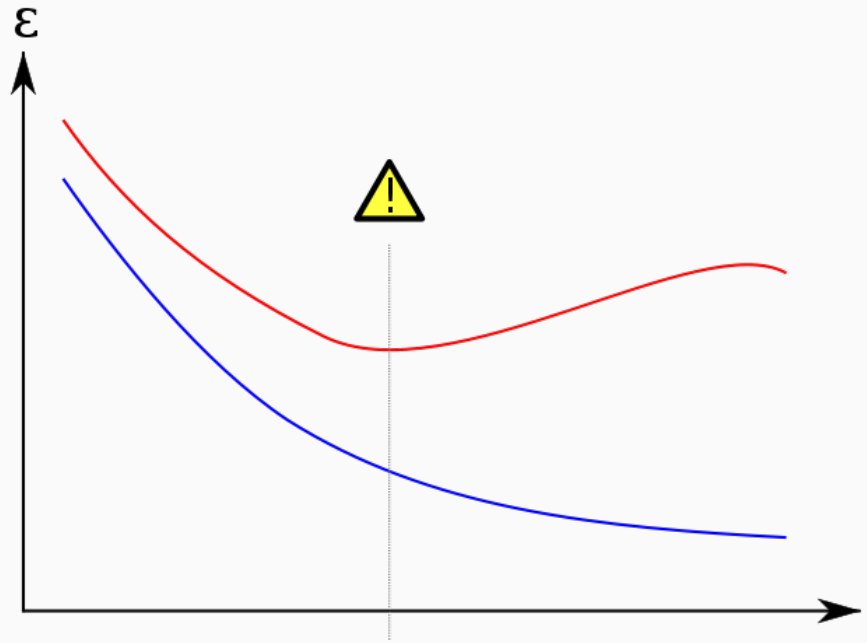
- У нейронных сетей *очень* много параметров.
- Регуляризация совершенно необходима.
- L_2 или L_1 регуляризация ($\lambda \sum_w w^2$ или $\lambda \sum_w |w|$) — это классический метод и в нейронных сетях, *weight decay*.
- Очень легко добавить: ещё одно слагаемое в целевую функцию, и иногда всё ещё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

Регуляризация в нейронных сетях

- 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.



Регуляризация в нейронных сетях

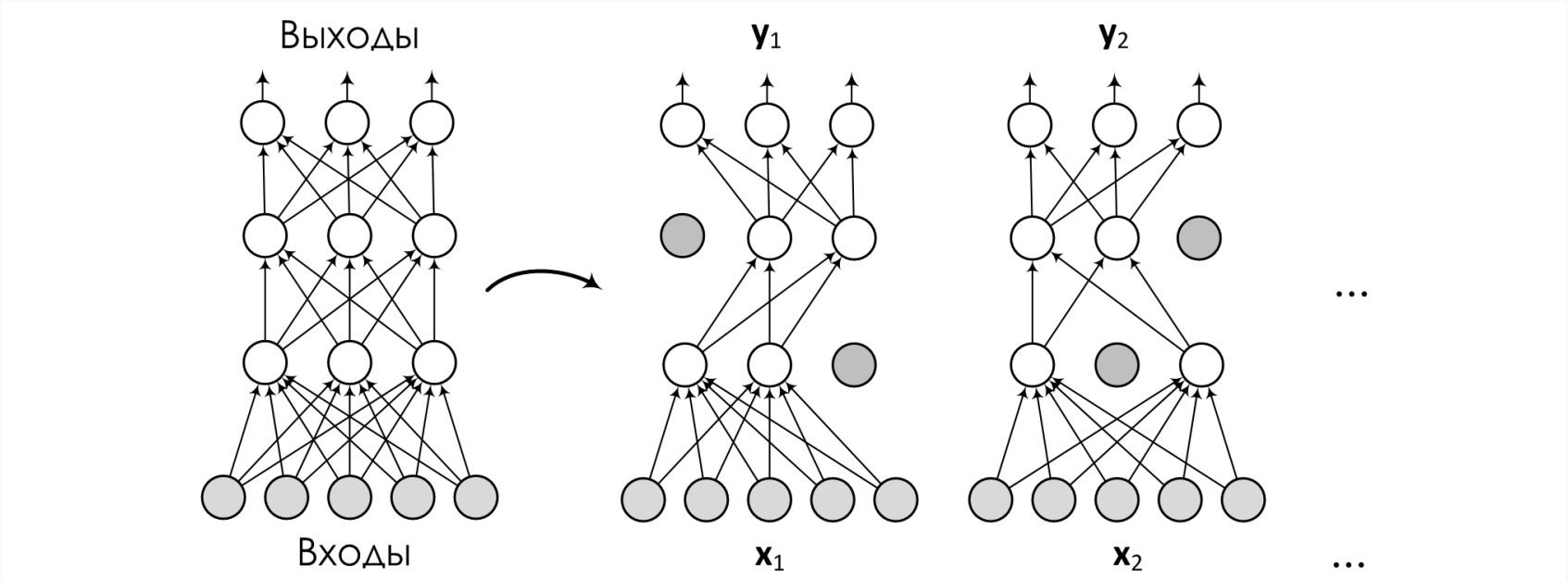
- Третий способ – max-norm constraint.
- Давайте искусственно ограничим норму вектора весов каждого нейрона:

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

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

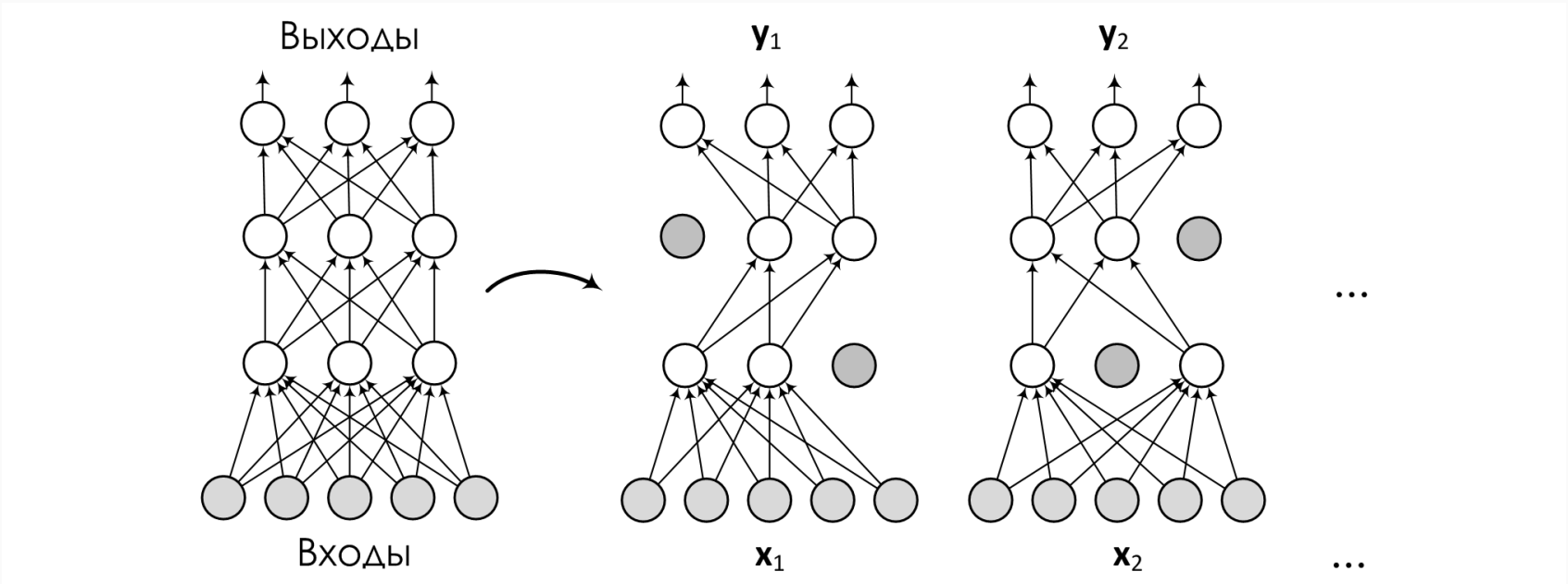
Дропаут

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



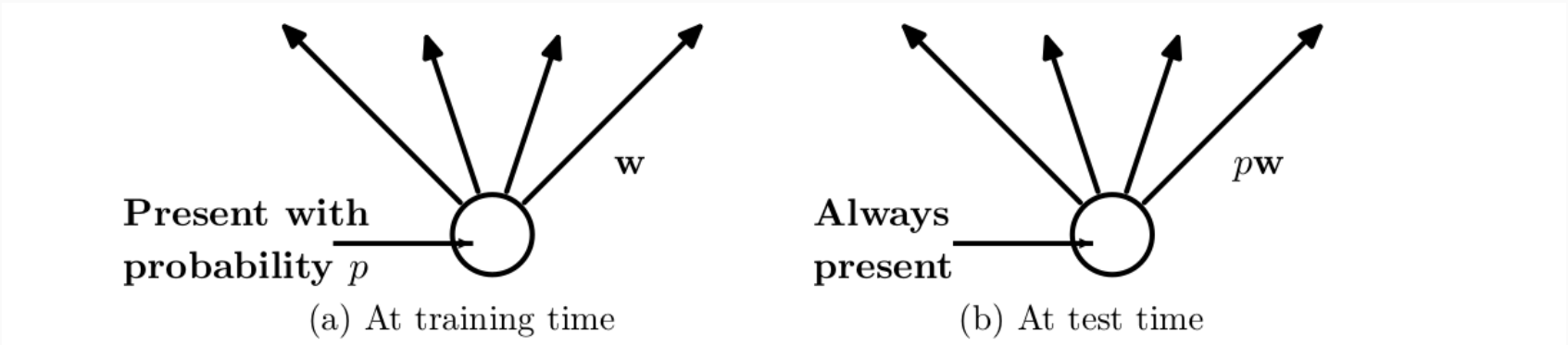
Дропаут

- 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?



Дропаут

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

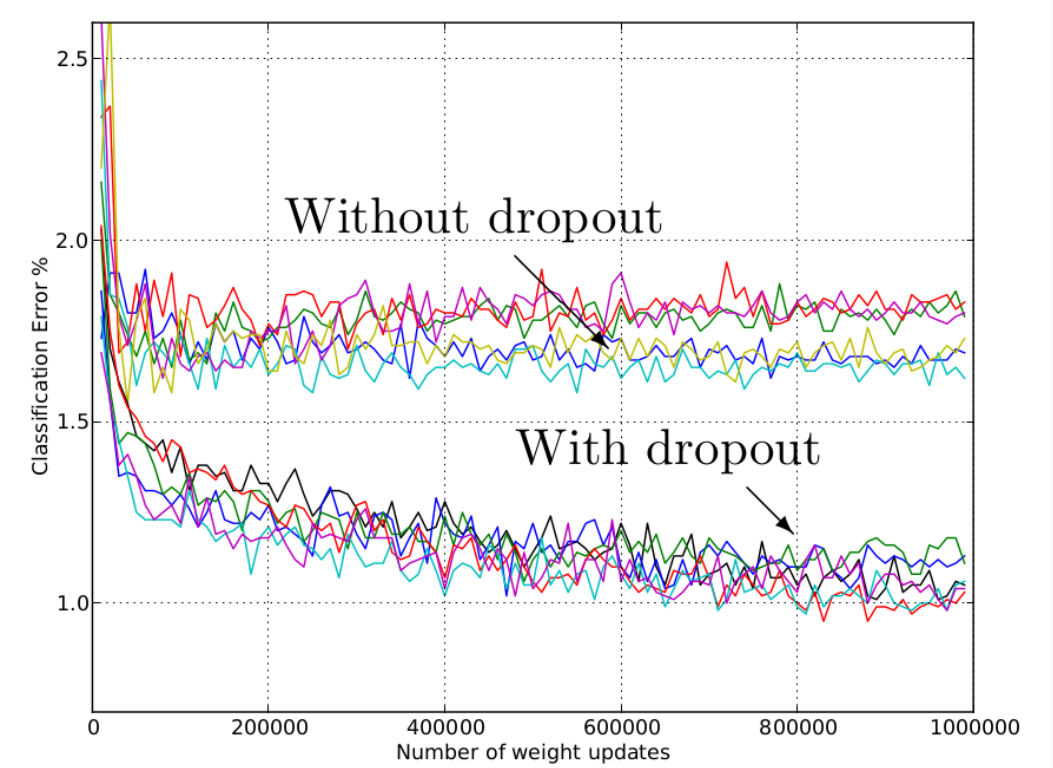


(a) At training time (b) At test time

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

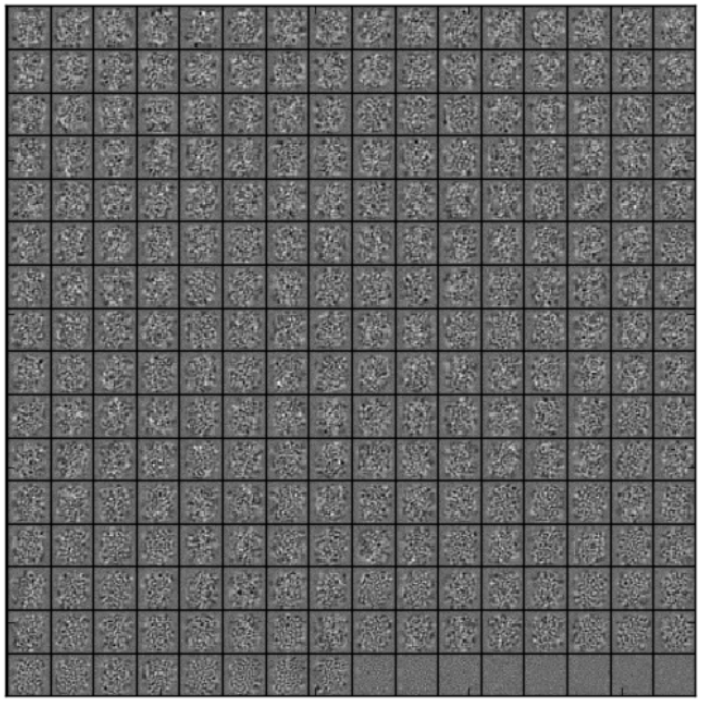
Дропаут

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

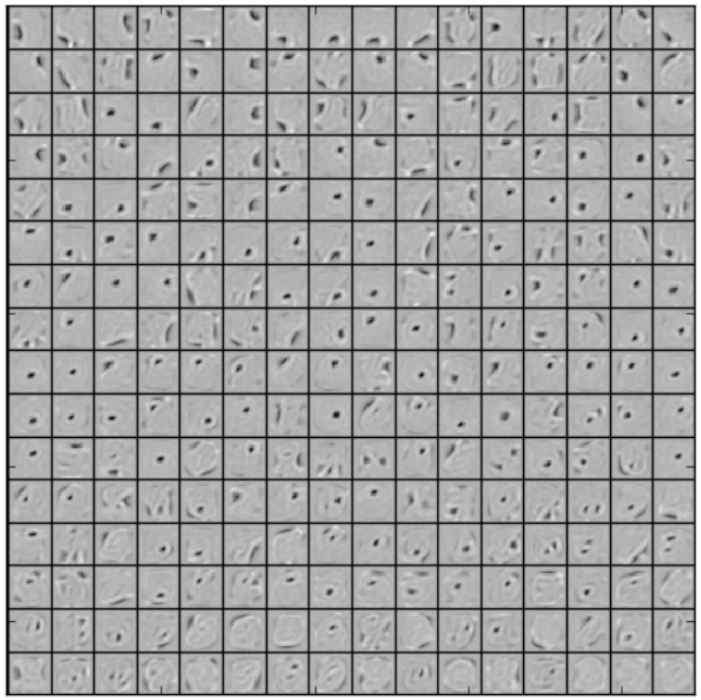


Дропаут

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



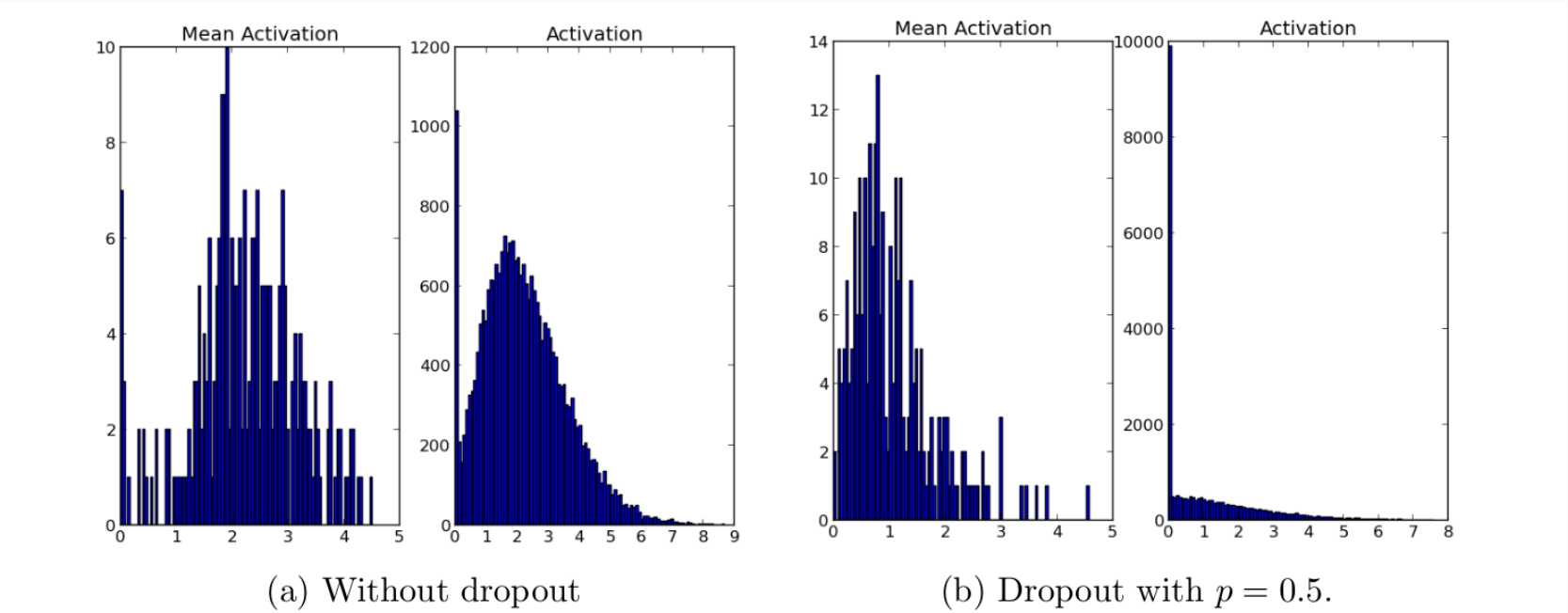
(a) Without dropout



(b) Dropout with $p = 0.5$.

Дропаут

- Аналогично, и разреженность появляется.



(a) Without dropout

(b) Dropout with $p = 0.5$.

Дропаут

- 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.

Дропаут

- 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

Дропаут

- 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.

Дропаут

- 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.

Вывод

- 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.

Спасибо за внимание!