DEEP LEARNING II: ГРАДИЕНТНЫЙ СПУСК

Сергей Николенко

НИУ ВШЭ — Санкт-Петербург
20 октября 2017 г.

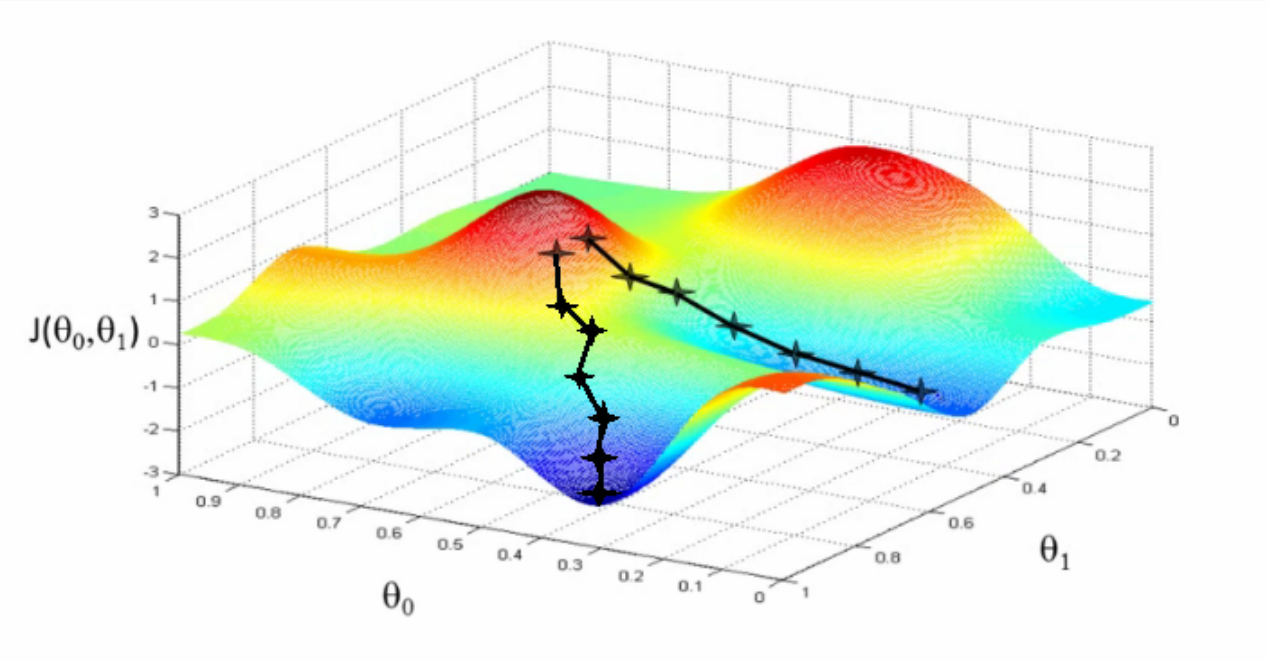
Random facts:

- 20 октября 1671 г. Людовик XIV повелел всем холостякам Новой Франции (от Луизианы до Ньюфаундленда) жениться на специально присланных из Франции девушках, «дочерях короля»
- 20 октября 1720 г. был пойман, а 17 ноября повешен Джек Рэкхем, знаменитый пират, известный как Калико Джек; захваченных вместе с ним
- 20 октября 1982 г. в конце матча Кубка УЕФА между «Спартаком» и «Хаарлемом» на «Лужниках» произошла массовая давка, в которой погибли 66 болельщиков
- 20 октября 1986 г. в Куйбышеве разбился самолет Ту-134А; командир поспорил, что сможет посадить самолет вслепую, пользуясь только приборами, и проиграл; в катастрофе погибло 70 человек из 94

Градиентный спуск и графы вычислений

ГРАДИЕНТНЫЙ СПУСК

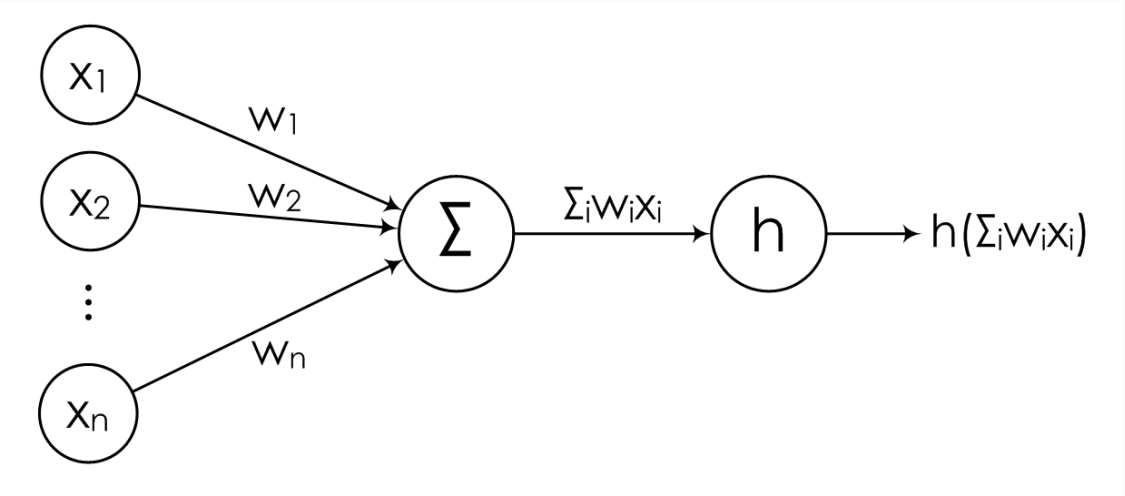
- Градиентный спуск — главный и фактически единственный способ оптимизации очень сложных функций.



- 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 ($h = \mathrm{id}$):

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

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

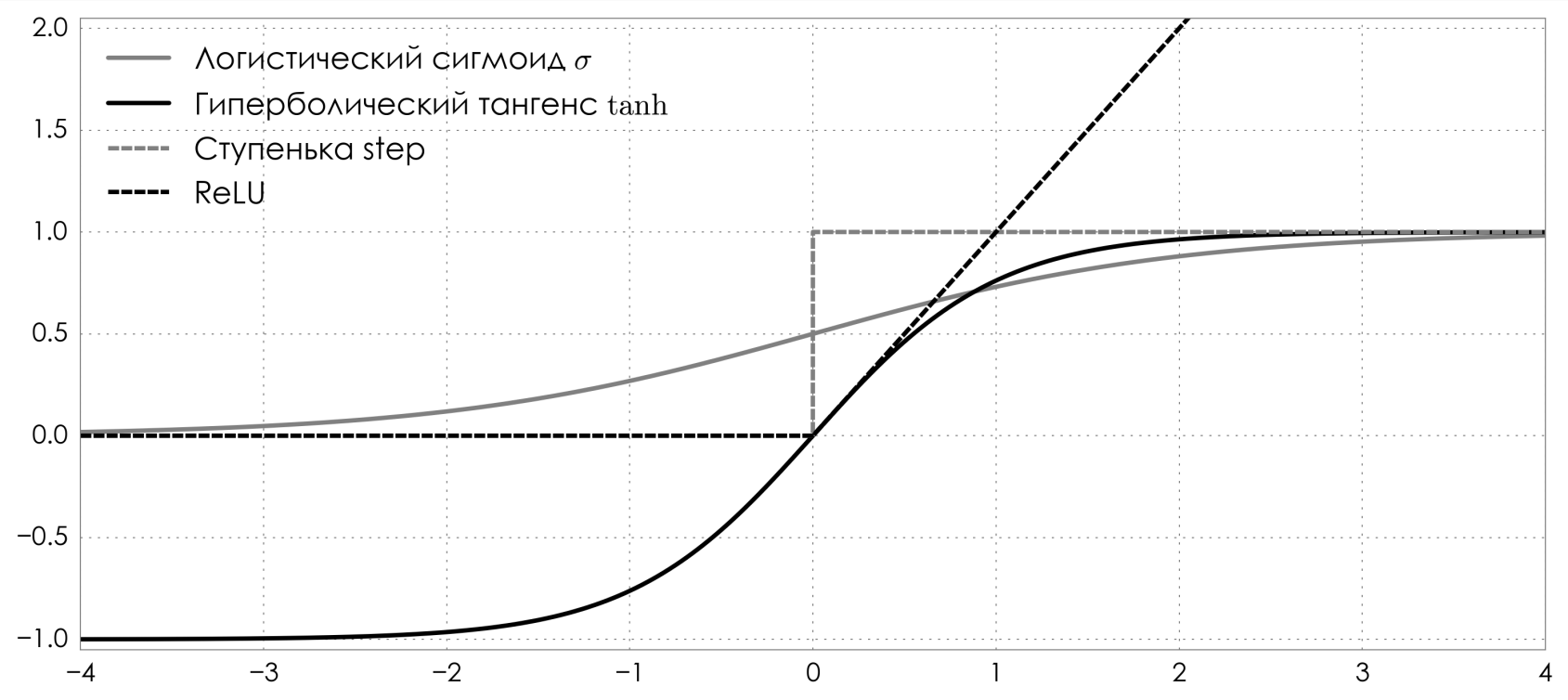
- Или, к примеру, можно делать классификацию, подставляя в качестве функции активации логистический сигмоид $h(x) = \sigma(x) = \frac{1}{1+e^{-x}}$:

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

- По сути это просто логистическая регрессия.

ПЕРЦЕПТРОН

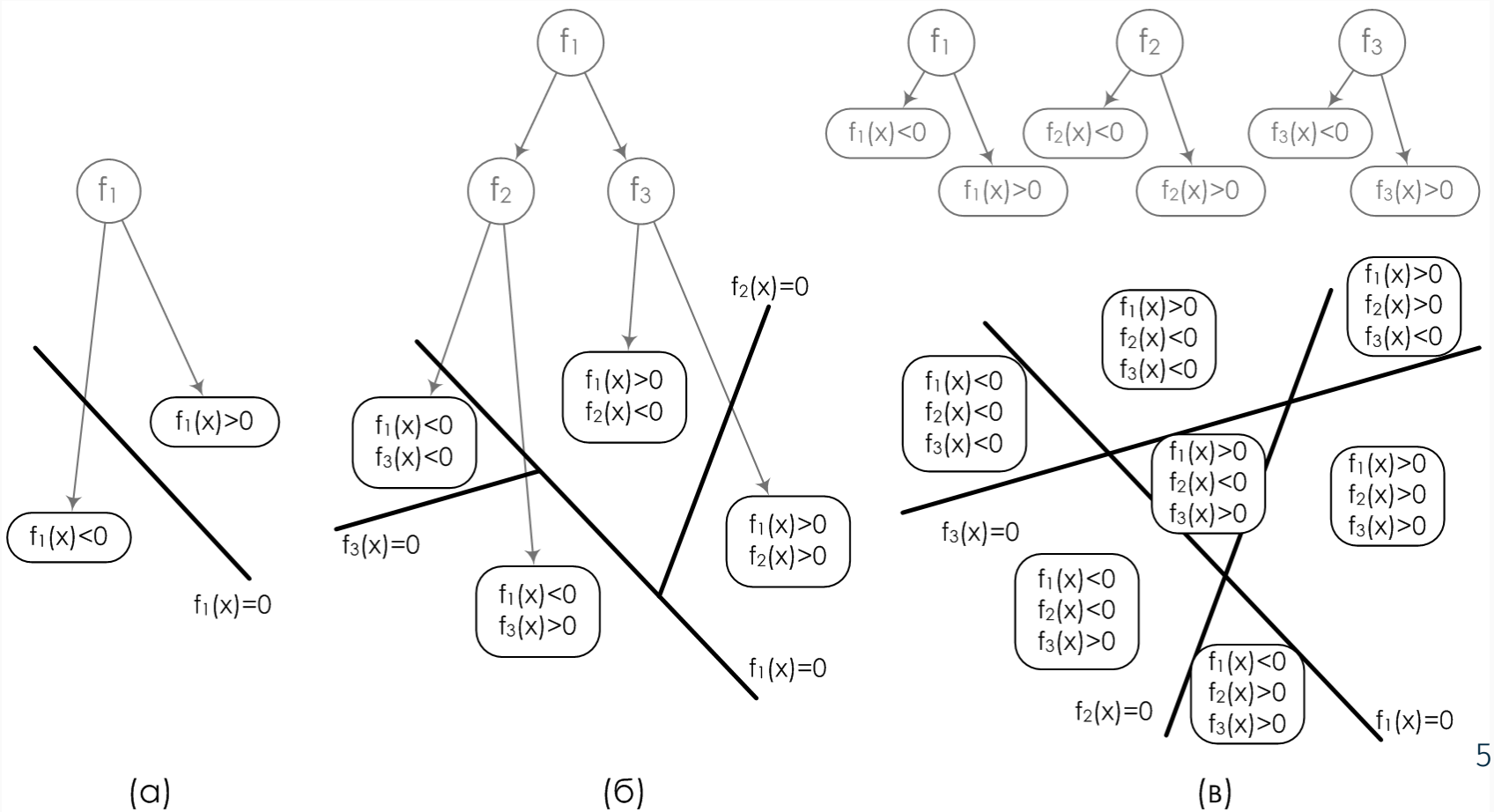
- Есть много разных нелинейных функций, которые можно использовать в перцептроне.



- Обычно главное – чтобы градиент хорошо проходил.

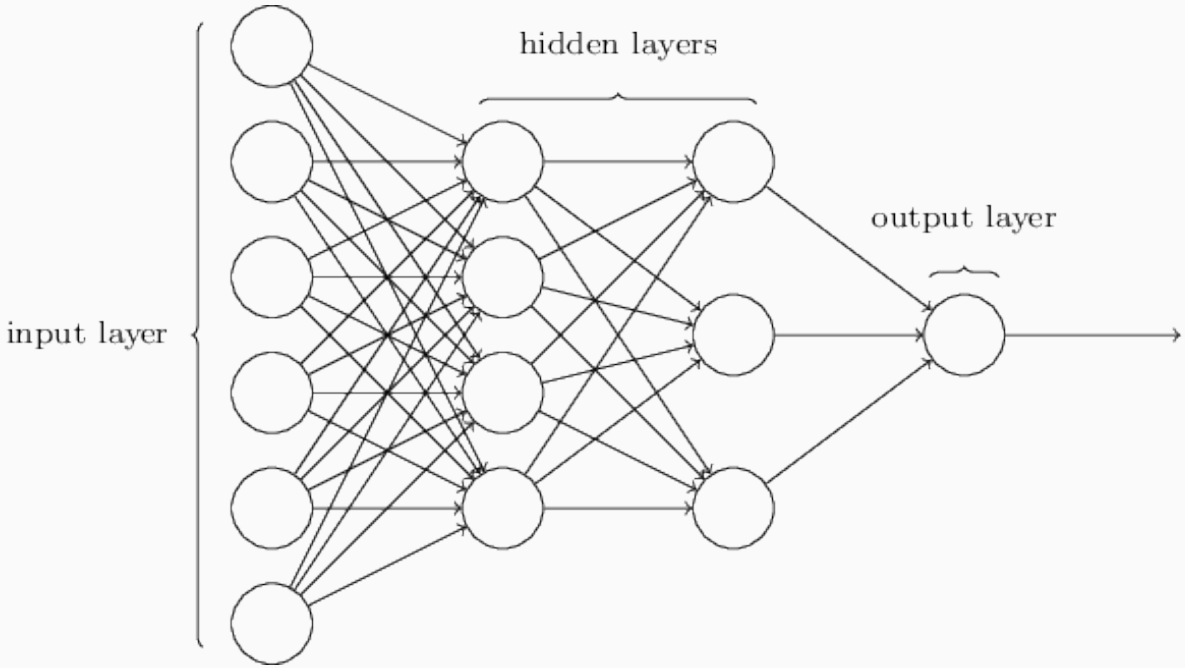
Объединяем перцептроны в сеть

- 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

- 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

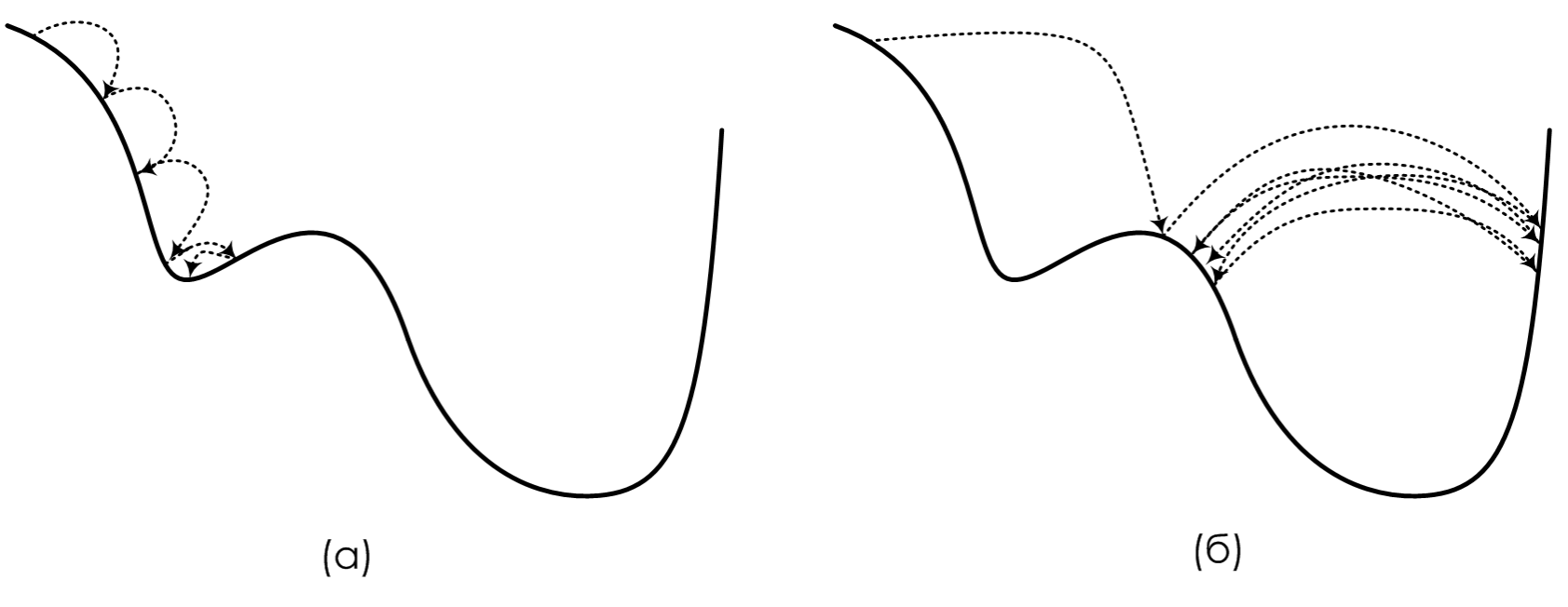
- Нет, конечно — *стохастический градиентный спуск* обновляет после каждого примера:

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

- 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

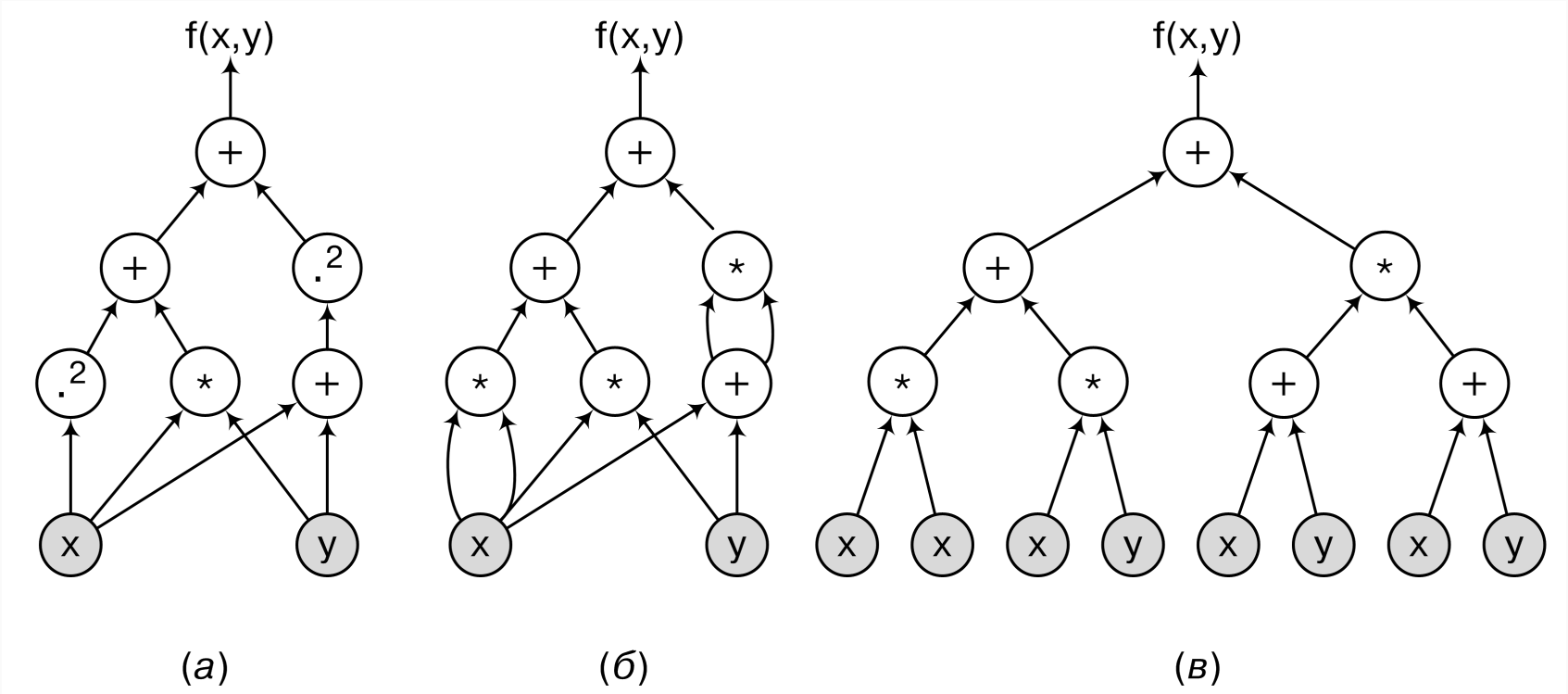
- Со скоростью обучения η масса проблем:



- 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, FROP 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:

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

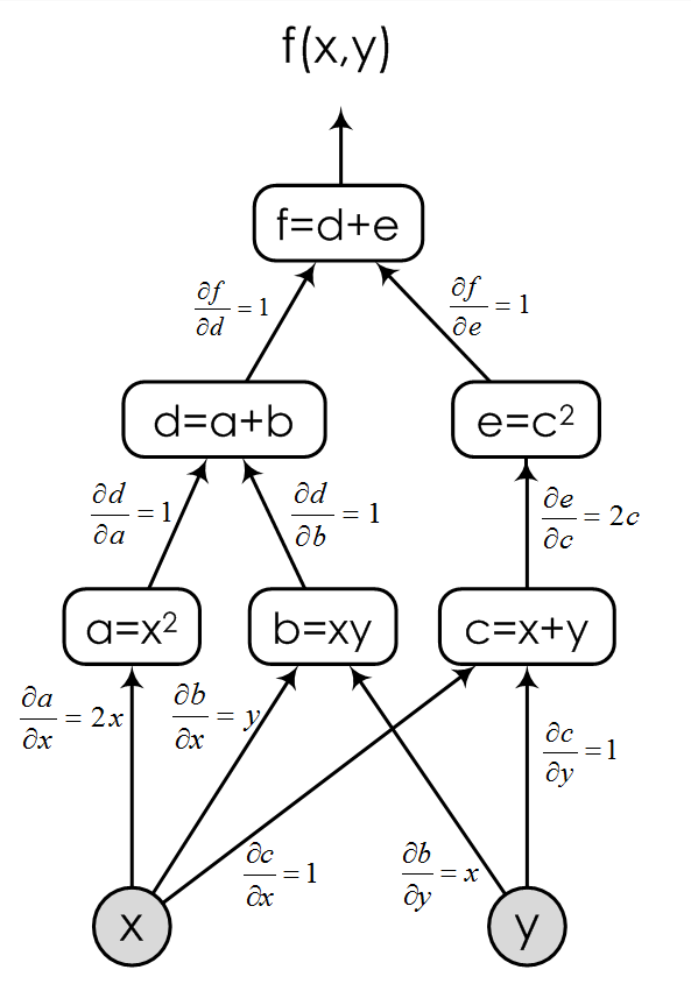
$$\nabla_{\mathbf{x}} f = \frac{\partial f}{\partial g_1}\nabla_{\mathbf{x}} g_1 + \dots + \frac{\partial f}{\partial g_k}\nabla_{\mathbf{x}} g_k = \sum_{i=1}^{k}\frac{\partial f}{\partial g_i}\nabla_{\mathbf{x}} g_i.$$

- Это матричное умножение на *матрицу Якоби*:

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

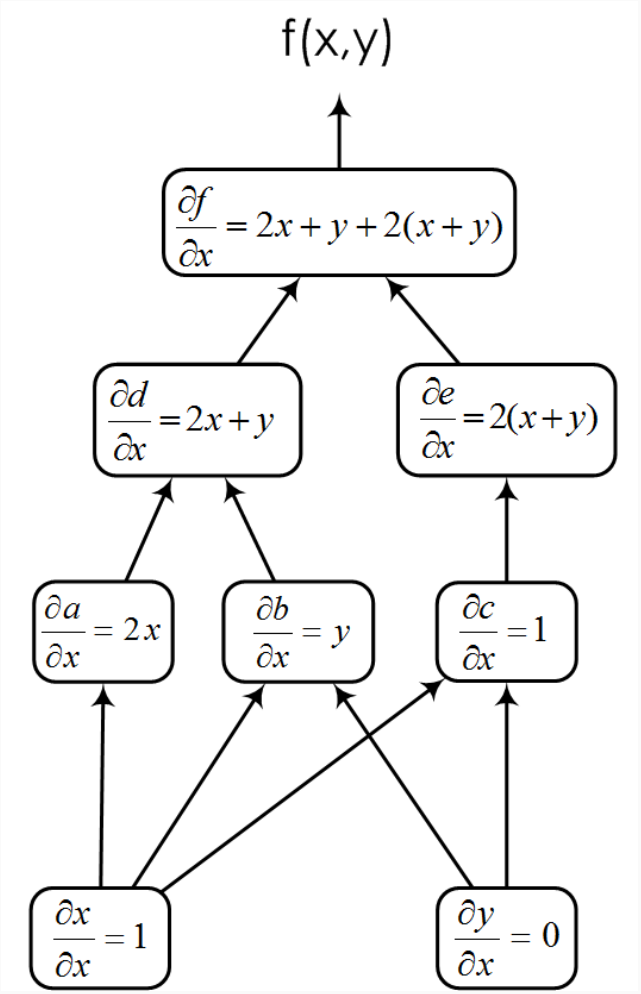
ГРАФ ВЫЧИСЛЕНИЙ, FROP AND BPROP

- Возвращаемся к примеру:



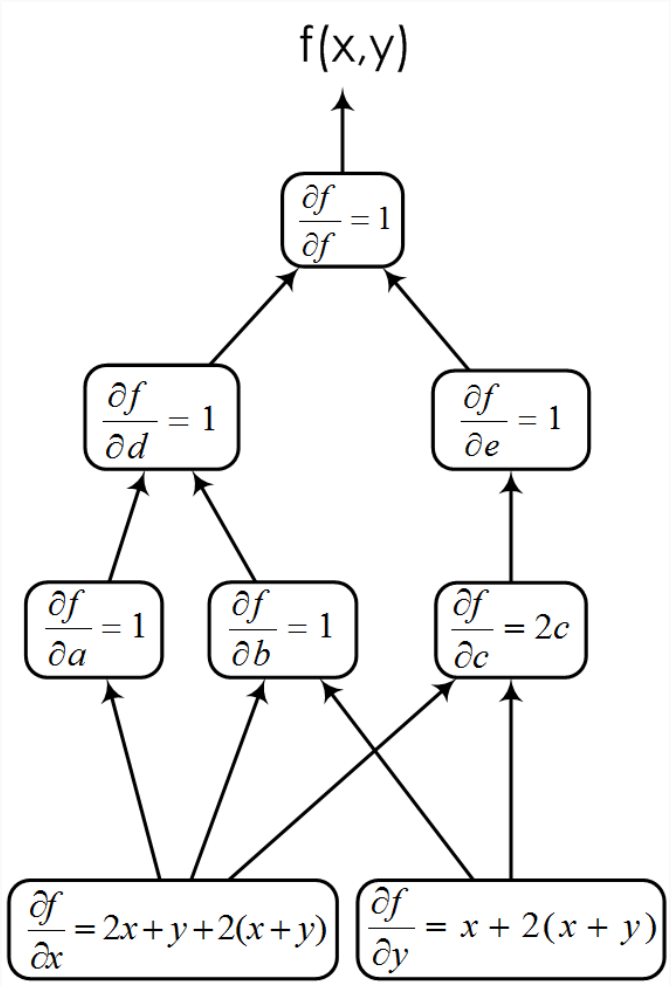
ГРАФ ВЫЧИСЛЕНИЙ, FROP AND BPROP

- 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 $\frac{\partial f}{\partial g} = \sum_{g' \in \text{Children}(g)} \frac{\partial f}{\partial g'} \frac{\partial g'}{\partial g}$.

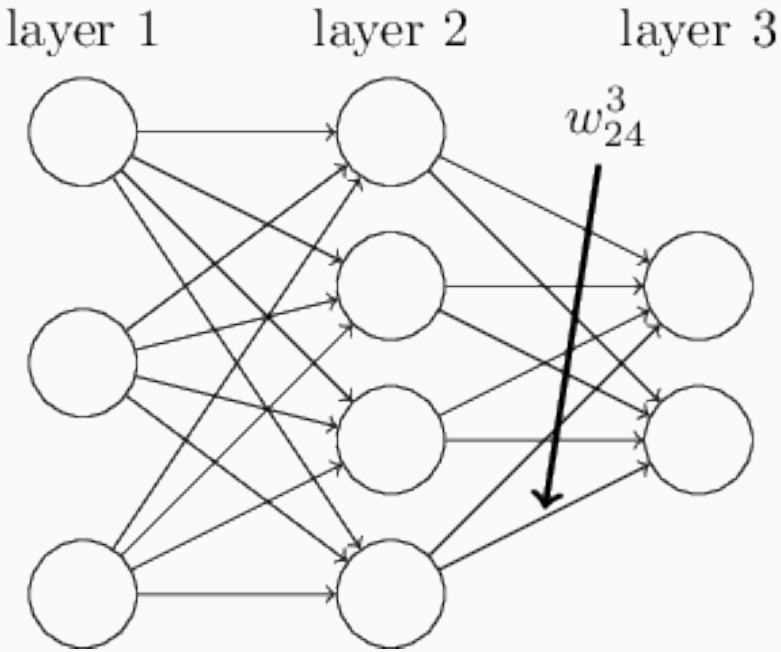


Граф вычислений, frop and bprop

- 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 *theano* и *TensorFlow* — это на самом деле библиотеки для *автоматического дифференцирования*, это их основная функция.
- И теперь мы можем реализовать массу “классических” моделей в *TensorFlow* 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:

$$\begin{aligned}
\delta_j^{(L)} &= \frac{\partial C}{\partial a^{(L)}} h'\left(z_j^{(L)}\right), \text{ т.е. } \delta^{(L)} = \nabla_{\mathbf{a}^{(L)}} C \odot h'\left(\mathbf{z}^{(L)}\right), \\
\delta^{(l)} &= \left((W^{(l+1)})^\top \delta^{(l+1)}\right) \odot h'\left(\mathbf{z}^{(l)}\right), \\
\frac{\partial C}{\partial b_j^{(l)}} &= \delta_j^{(l)}, \\
\frac{\partial C}{\partial w_{jk}^{(l)}} &= a_k^{(l-1)} \delta_j^{(l)}.
\end{aligned}$$

- Это всё операции, которые легко параллелизовать на GPU.

ГРАФ ВЫЧИСЛЕНИЙ, FROP 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).

О перекрестной энтропии

- Замечание о перекрестной энтропии:

$$L(\mathbf{w}) = \sum_{n=1}^{N} \left[t_n \ln a^{(L)} + (1 - t_n) \ln(1 - a^{(L)}) \right].$$

- Здесь $a^{(L)} = \sigma(z^{(L)})$, а $z^{(L)} = (\mathbf{w}^{(L)})^\top \mathbf{x} + b^{(L)}$.
- Заметим, что $\sigma'(x) = \sigma(x)(1 - \sigma(x))$. То есть

$$\frac{\partial L}{\partial w_j} = -\sum_{n=1}^{N} \left[\frac{t_n}{\sigma(z^{(L)})} - \frac{1 - t_n}{1 - \sigma(z^{(L)})} \right] \sigma'(z^{(L)}) \mathbf{x}_n.$$

- И всё упрощается, и получается

$$\frac{\partial L}{\partial w_j} = \sum_{n} (\sigma(z^{(L)}) - t_n) \mathbf{x}_n,$$

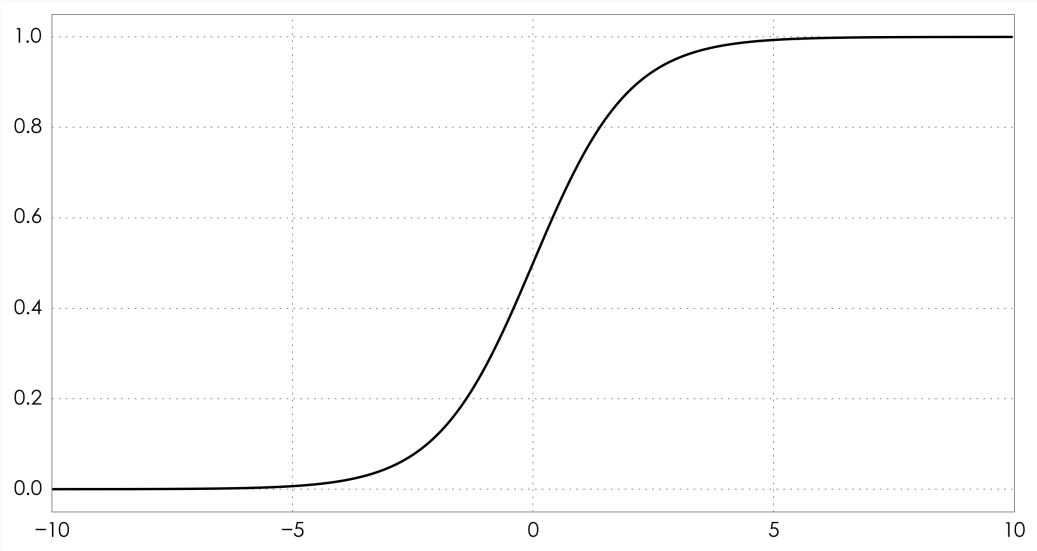
т.е. чем больше ошибка, тем быстрее будет нейрон обучаться.

О ПЕРЕКРЕСТНОЙ ЭНТРОПИИ

- А что было бы для квадратичной ошибки $L(\mathbf{w}) = \sum_{n=1}^{N} \left(t_n - a^{(L)}\right)^2$?
- Было бы

$$\frac{\partial L}{\partial w_j} = \sum_n (\sigma(z^{(L)}) - t_n)\sigma'(z^{(L)})\mathbf{x}_n,$$

и это нехорошо из-за насыщений.



РЕГУЛЯРИЗАЦИЯ В НЕЙРОННЫХ СЕТЯХ

Регуляризация в нейронных сетях

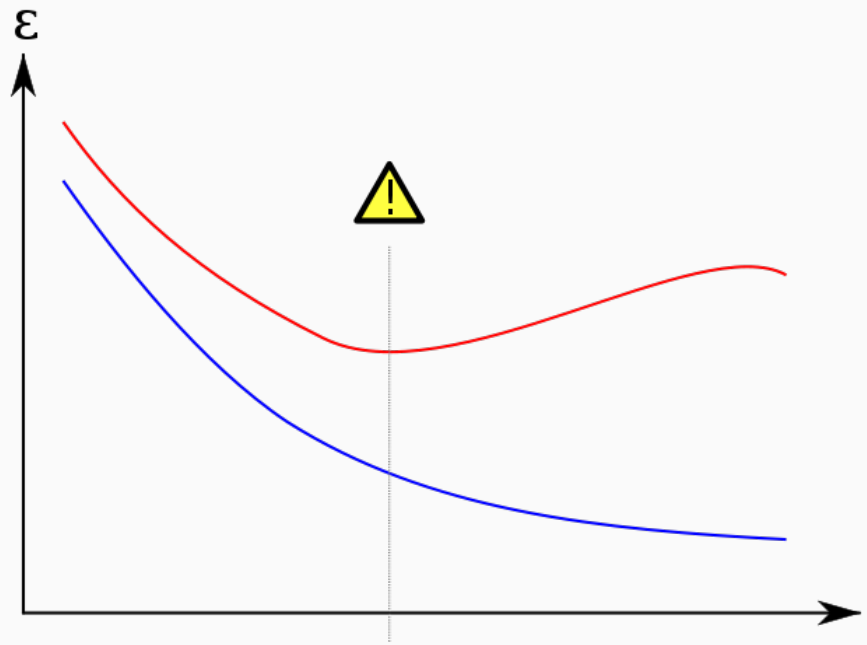
- У нейронных сетей *очень* много параметров.
- Регуляризация совершенно необходима.
- L_2 или L_1 регуляризация ($\lambda \sum_w w^2$ или $\lambda \sum_w |w|$) — это классический метод и в нейронных сетях, *weight decay*.
- Очень легко добавить: ещё одно слагаемое в целевую функцию, и иногда всё ещё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

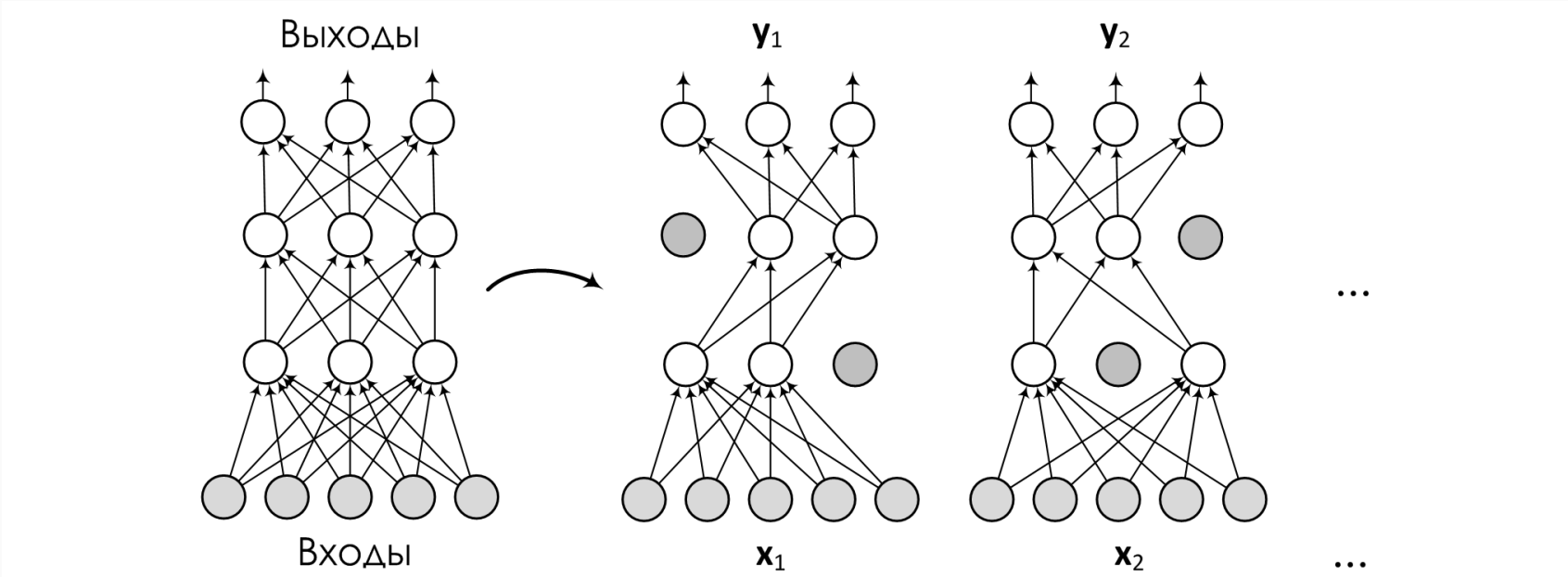
- Третий способ – max-norm constraint.
- Давайте искусственно ограничим норму вектора весов каждого нейрона:

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

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

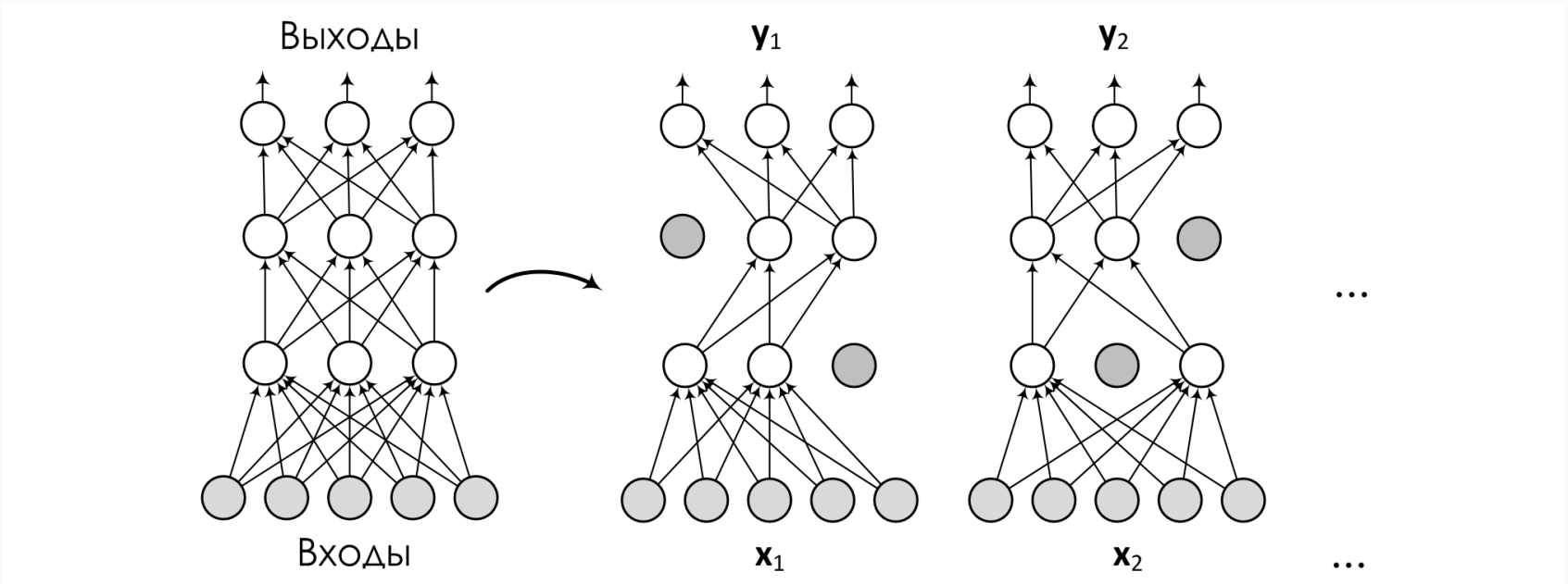
ДРОПАУТ

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



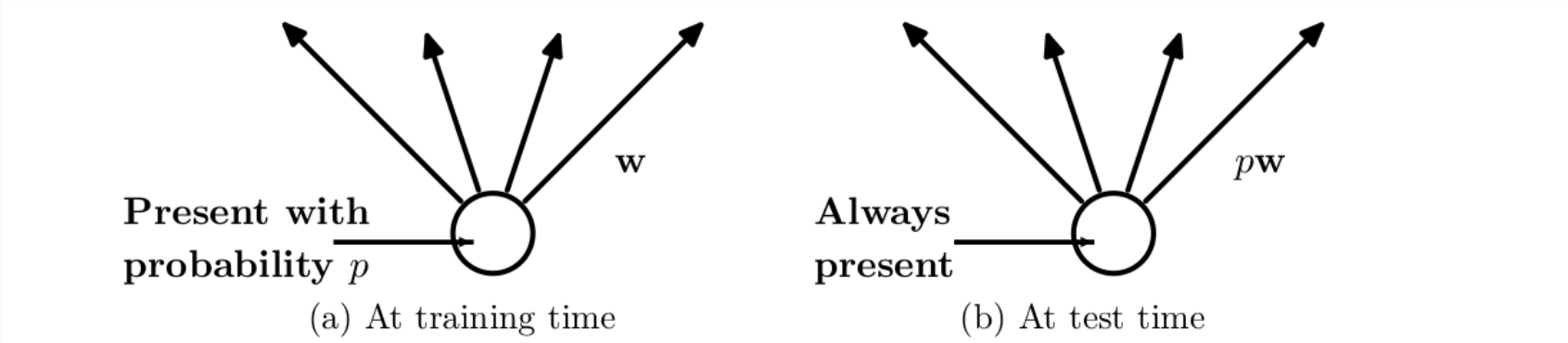
ДРОПАУТ

- 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!

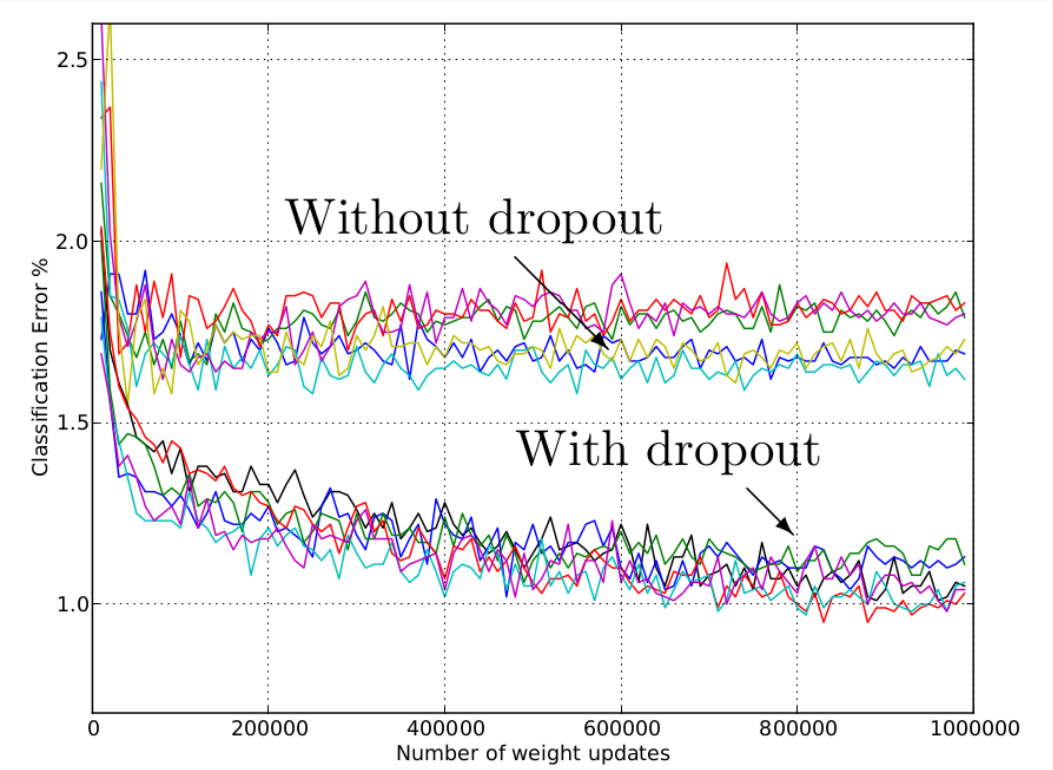


(a) At training time (b) At test time

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

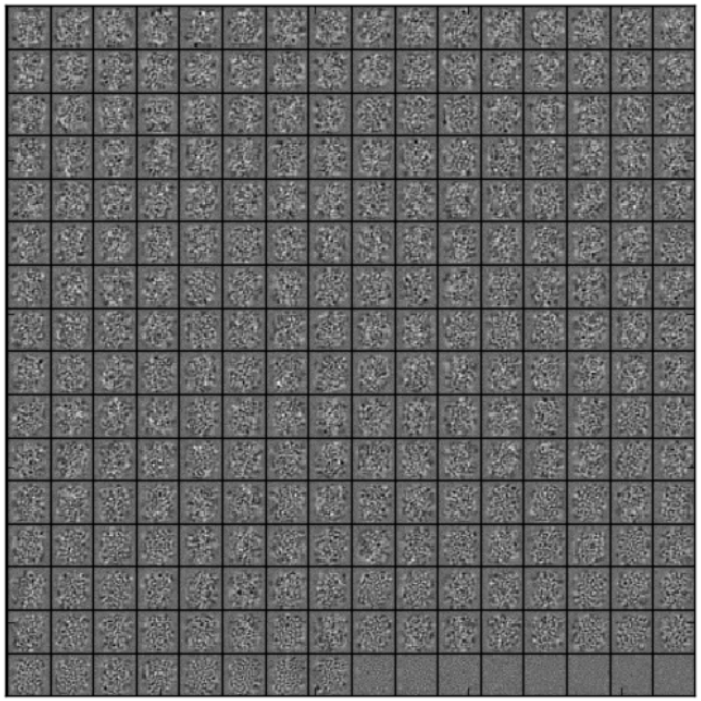
ДРОПАУТ

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

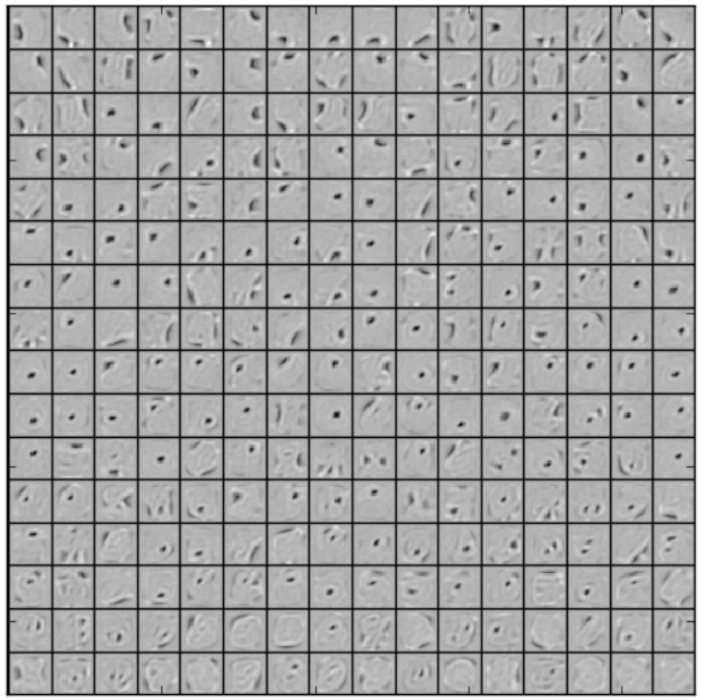


ДРОПАУТ

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



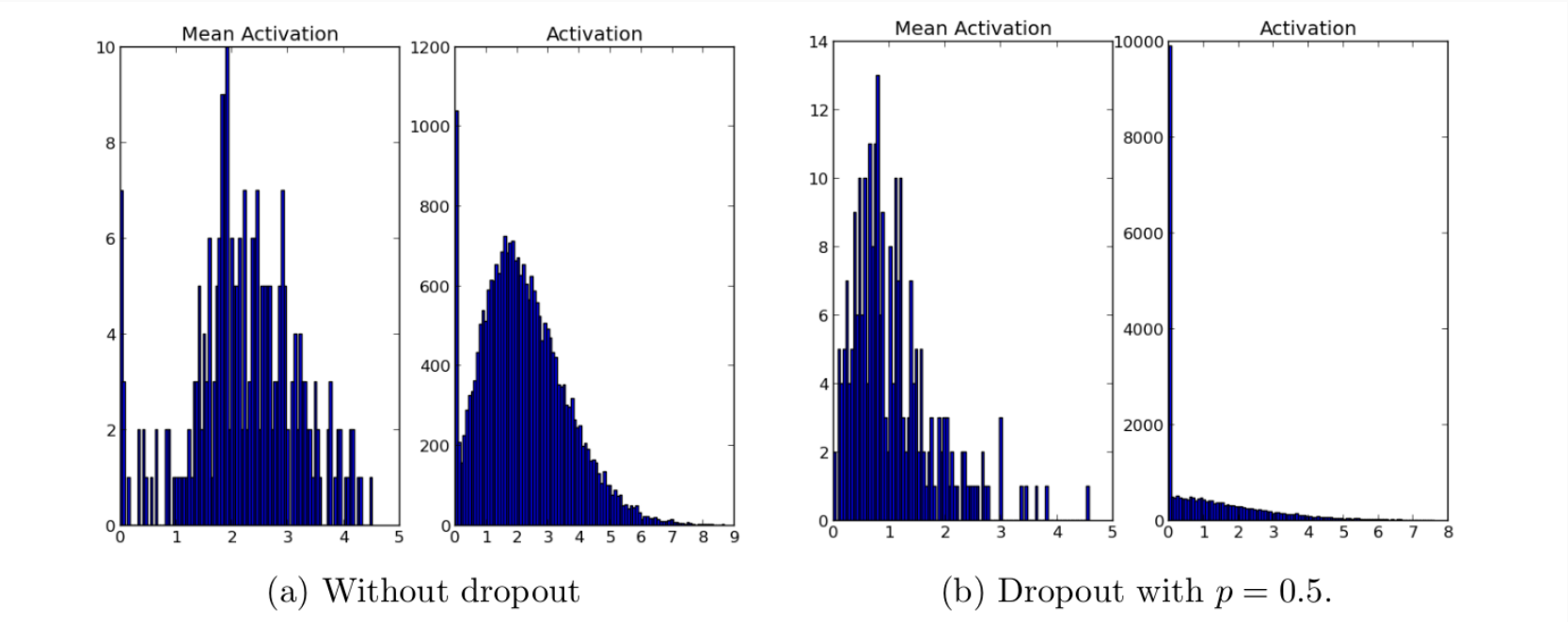
(a) Without dropout



(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

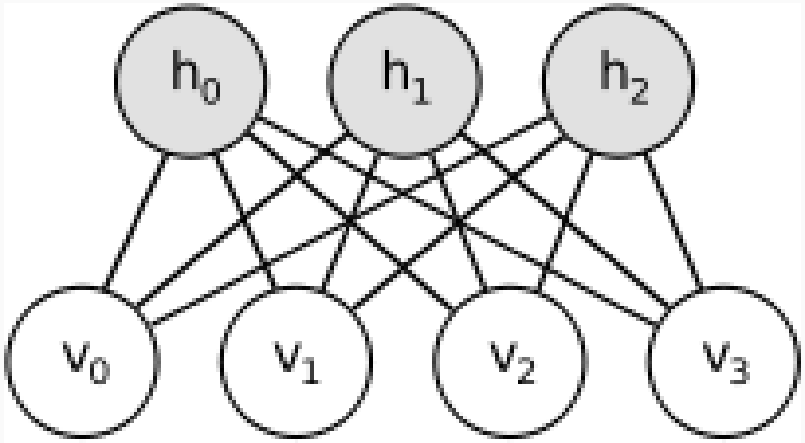
- Итого получается, что dropout – это очень крутой метод.

- 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.

Инициализация весов

ПРЕДОБУЧЕНИЕ БЕЗ УЧИТЕЛЯ

- 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):



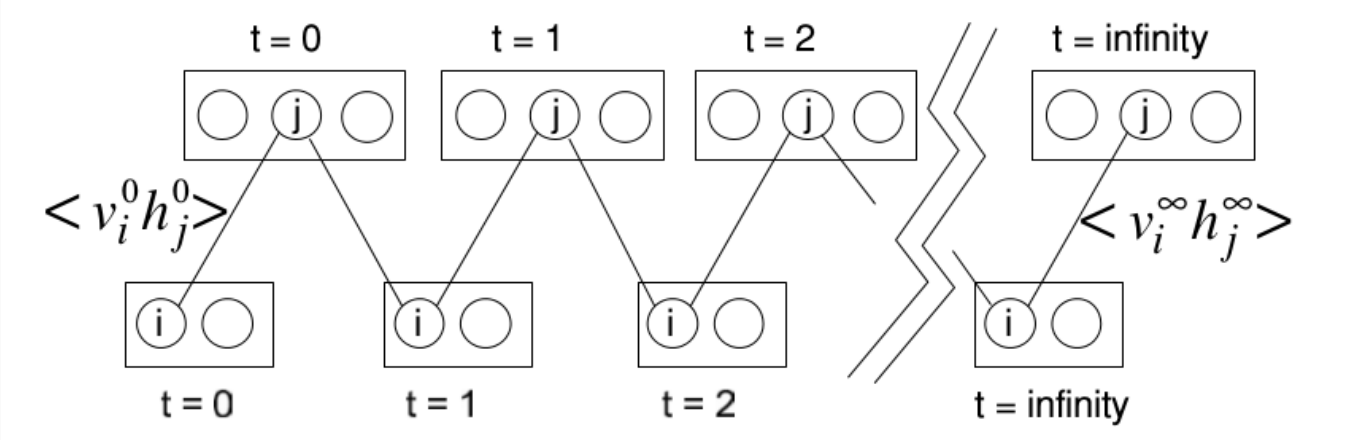
ПРЕДОБУЧЕНИЕ БЕЗ УЧИТЕЛЯ

- Это ненаправленная графическая модель, задающая распределение

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

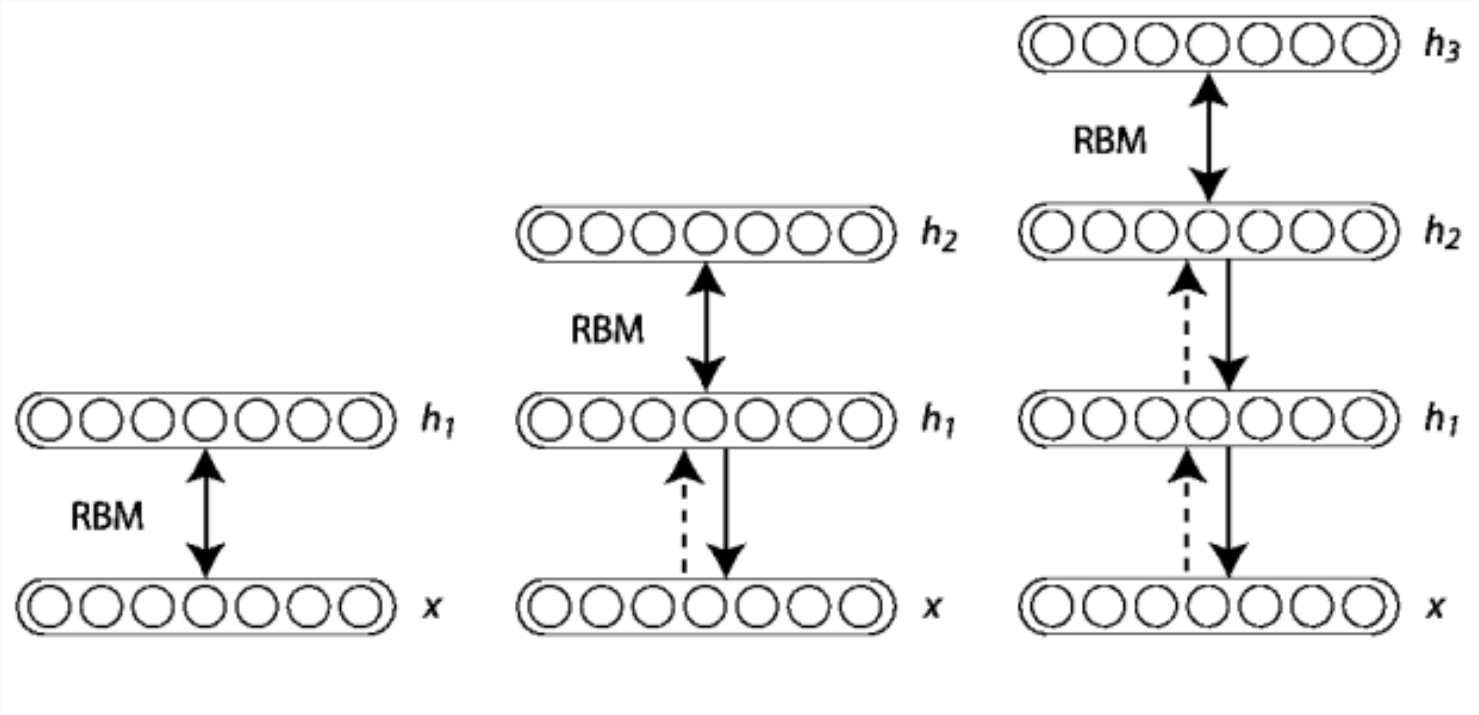
$$E(\mathbf{v}, h) = -\mathbf{b}^\top \mathbf{v} - \mathbf{c}^\top h - h^\top W \mathbf{v}.$$

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



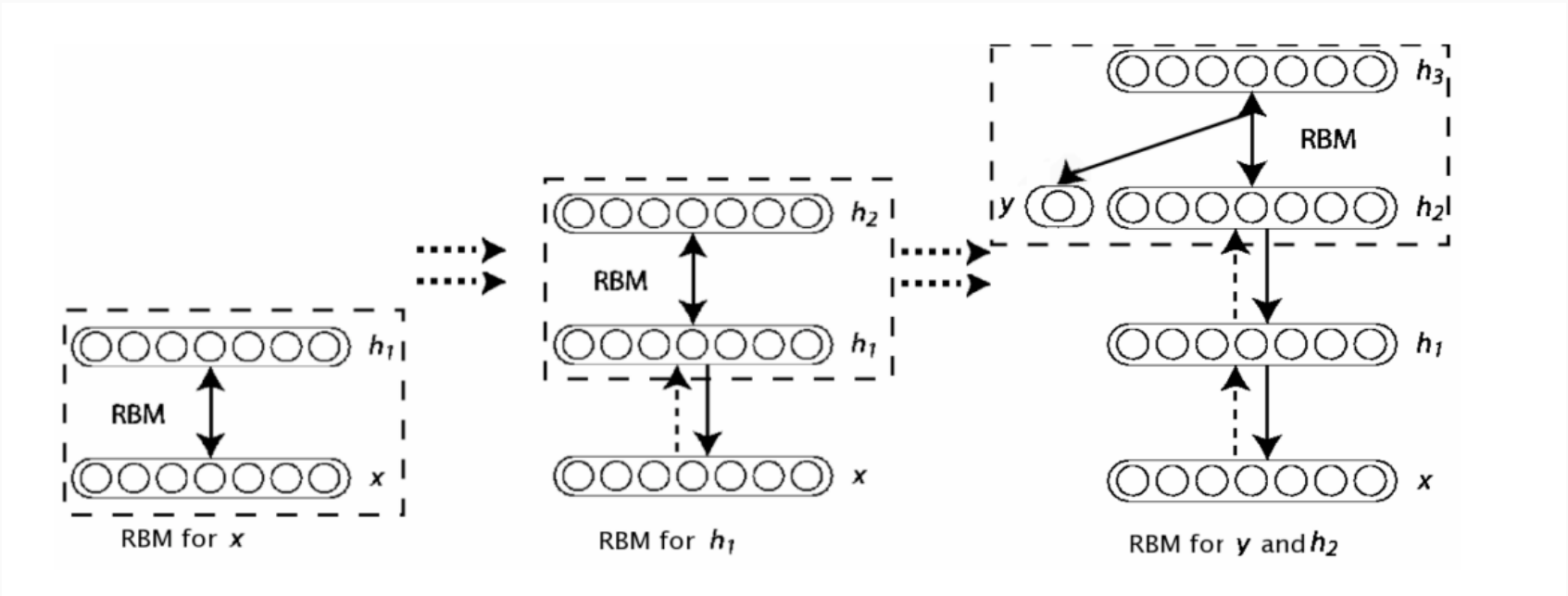
ПРЕДОБУЧЕНИЕ БЕЗ УЧИТЕЛЯ

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



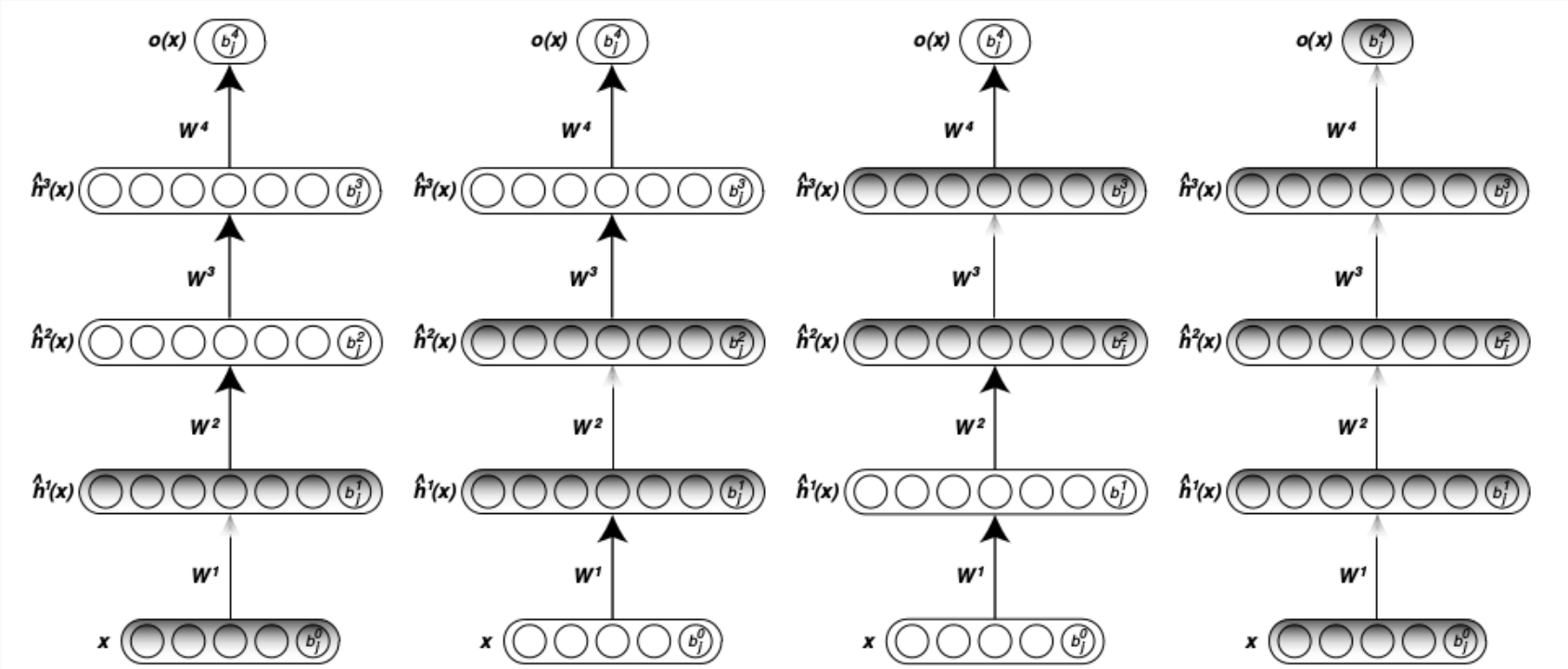
ПРЕДОБУЧЕНИЕ БЕЗ УЧИТЕЛЯ

- И вывод можно вести последовательно, уровень за уровнем:



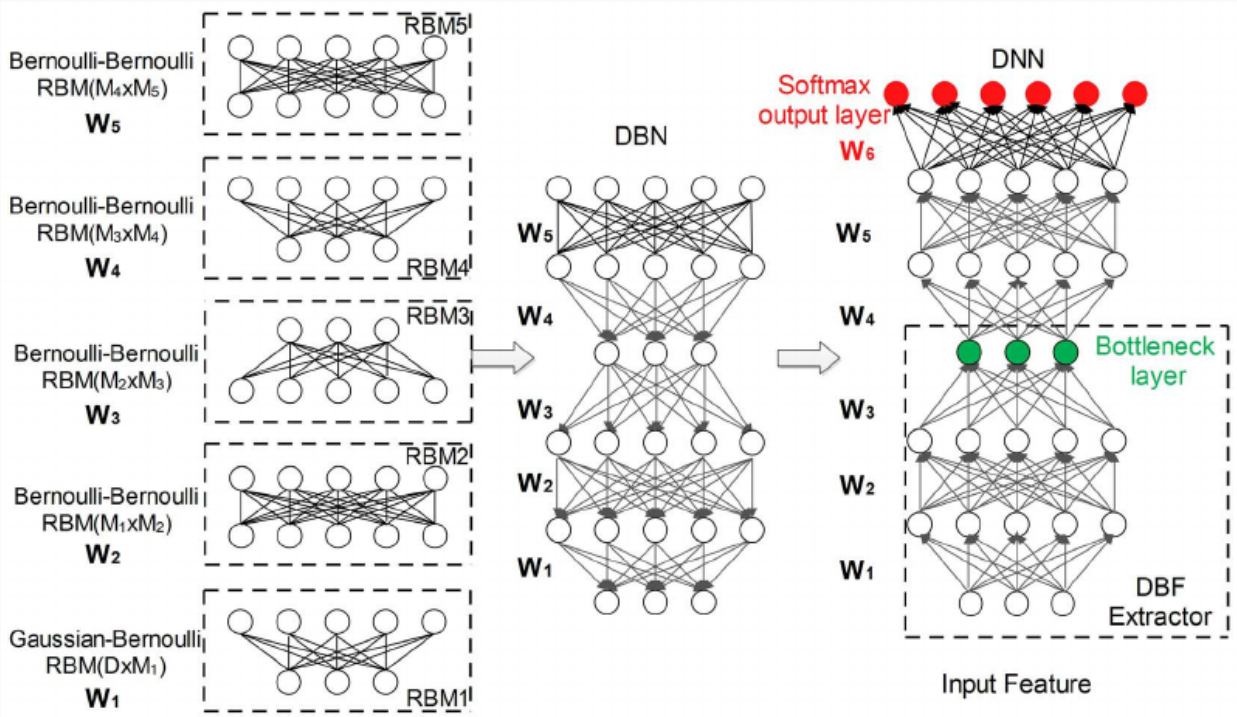
ПРЕДОБУЧЕНИЕ БЕЗ УЧИТЕЛЯ

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



ПРЕДОБУЧЕНИЕ БЕЗ УЧИТЕЛЯ

- Этот подход привёл к прорыву в распознавании речи.



ПРЕДОБУЧЕНИЕ БЕЗ УЧИТЕЛЯ

- 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.

ИНИЦИАЛИЗАЦИЯ ХАВЬЕРА

- *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

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

ИНИЦИАЛИЗАЦИЯ ХАВЬЕРА

- Его дисперсия равна

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

- Для нулевого среднего весов

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

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

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

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

ИНИЦИАЛИЗАЦИЯ ХАВЬЕРА

- 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

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

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

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

ИНИЦИАЛИЗАЦИЯ ХАВЬЕРА

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

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

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...

ИНИЦИАЛИЗАЦИЯ ХАВЬЕРА

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

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

- Мы теперь можем обнулить только второе слагаемое:

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

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

ИНИЦИАЛИЗАЦИЯ ХАВЬЕРА

- Мы теперь можем обнулить только второе слагаемое:

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

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

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

ИНИЦИАЛИЗАЦИЯ ХАВЬЕРА

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

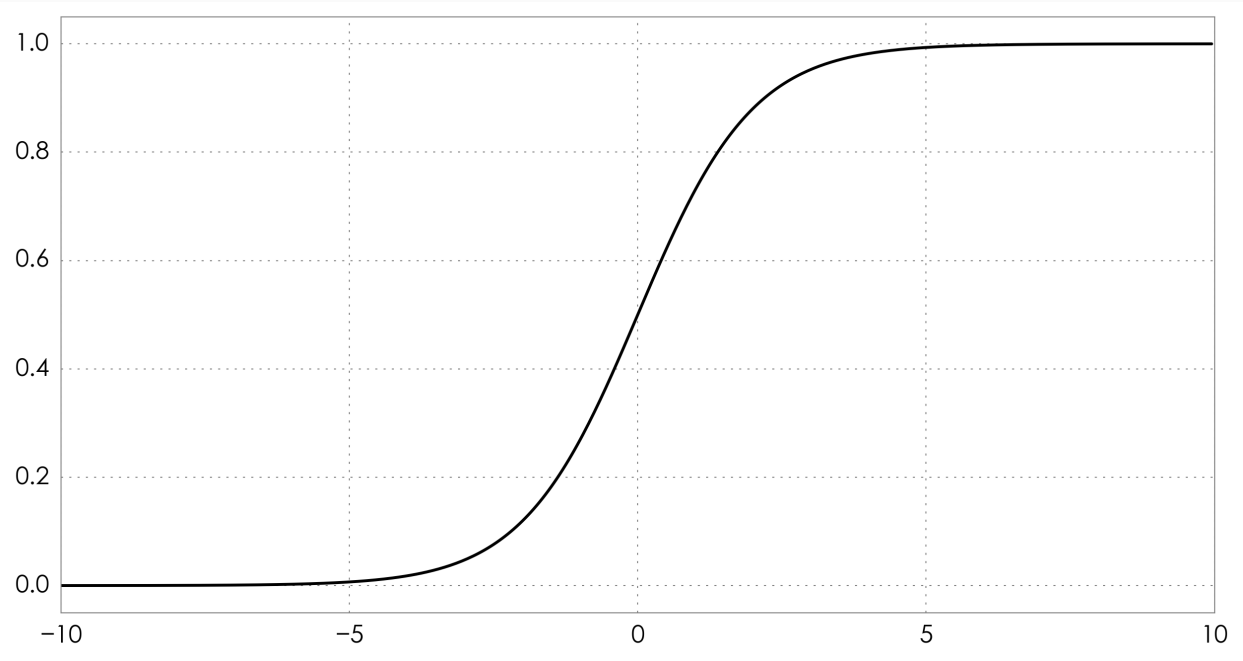
$$\text{Var}\left[w_i\right] = 2/n_{\text{in}}^{(l)}.$$

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

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

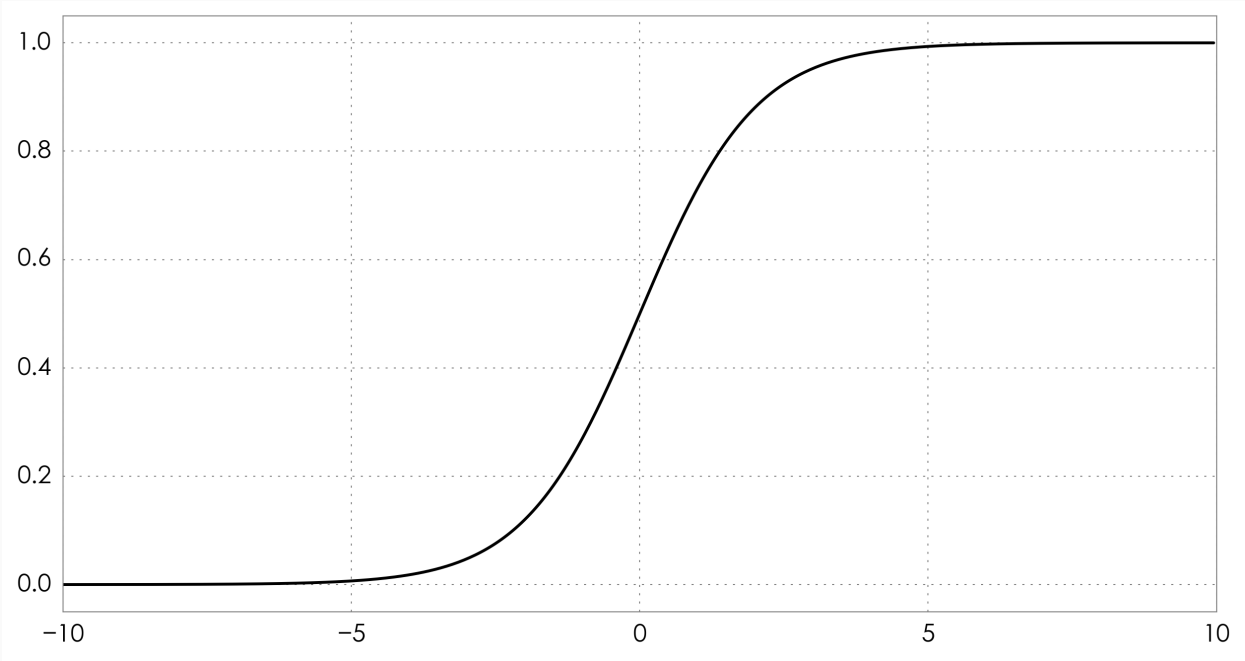
О сигмоидах

- 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.



О сигмоидах

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



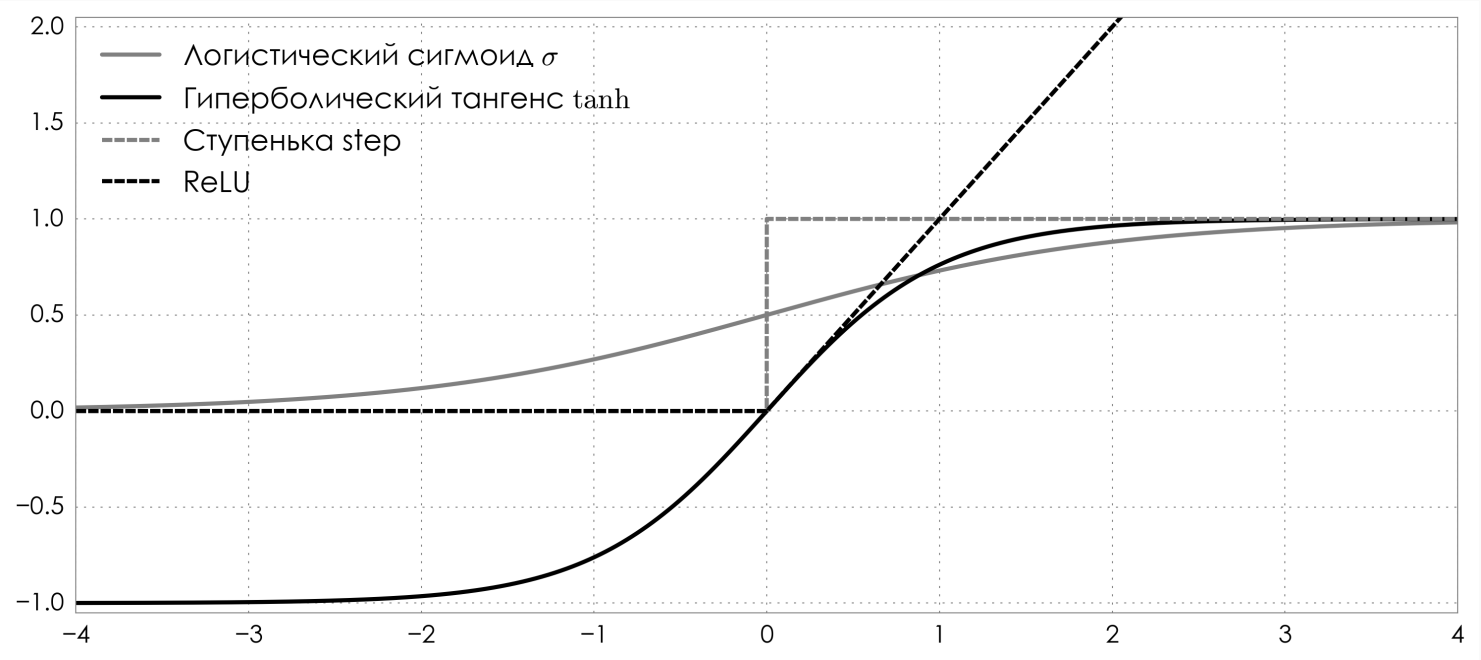
- Но ведь другие тоже насыщаются? В чём разница?

О сигмоидах

- 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 $\mathrm{softmax}$.
- Когда мы начинаем оптимизировать сложную функцию потерь, поначалу выходы 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 Wh, которое поначалу скорее шум, чем сигнал.

О сигмоидах

- 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 $\sigma(x) \to 0$ будет и $\sigma'(x) \to 0$.
- А у $\tanh$ наоборот: когда $\tanh(x) \to 0$, $\tanh'(x)$ максимальна.



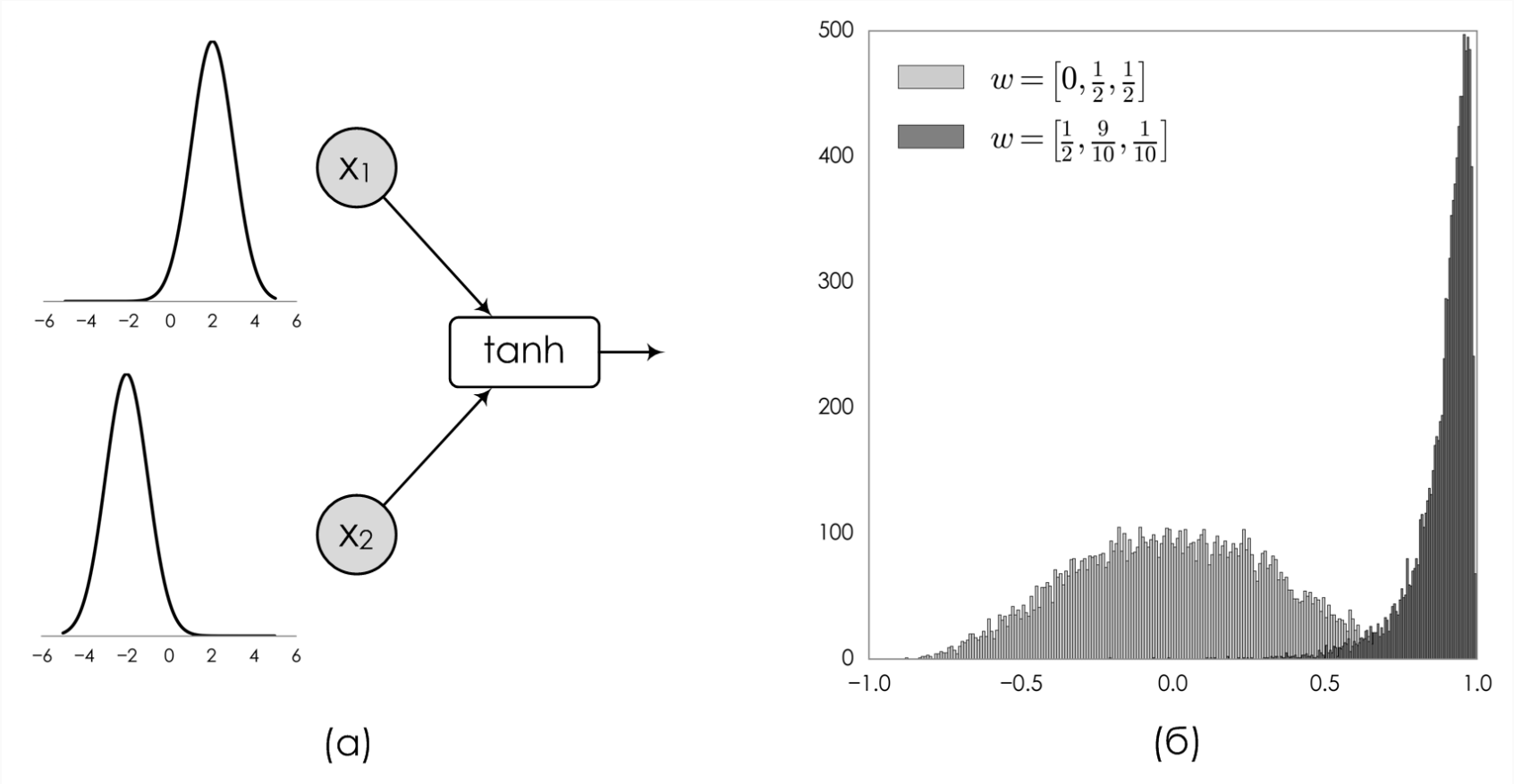
Нормализация по мини-батчам

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?

Нормализация по мини-батчам

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

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

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

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

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

Нормализация по мини-батчам

- Можно пытаться добавить нормализацию как отдельный слой:

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

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

$$\mathrm{Cov}[\mathbf{x}] = \mathbb{E}_{\mathbf{x} \in \mathcal{X}} \left[\mathbf{x}\mathbf{x}^\top\right] - \mathbb{E}\left[\mathbf{x}\right] \mathbb{E}\left[\mathbf{x}\right]^\top.$$

- Это точно не сработает.

Нормализация по мини-батчам

- 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

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

  где статистики подсчитаны по текущему мини-батчу.
- Ещё одна проблема: теперь пропадают нелинейности!
- Например, σ теперь практически всегда близка к линейной.

Нормализация по мини-батчам

- 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):

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

- 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

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

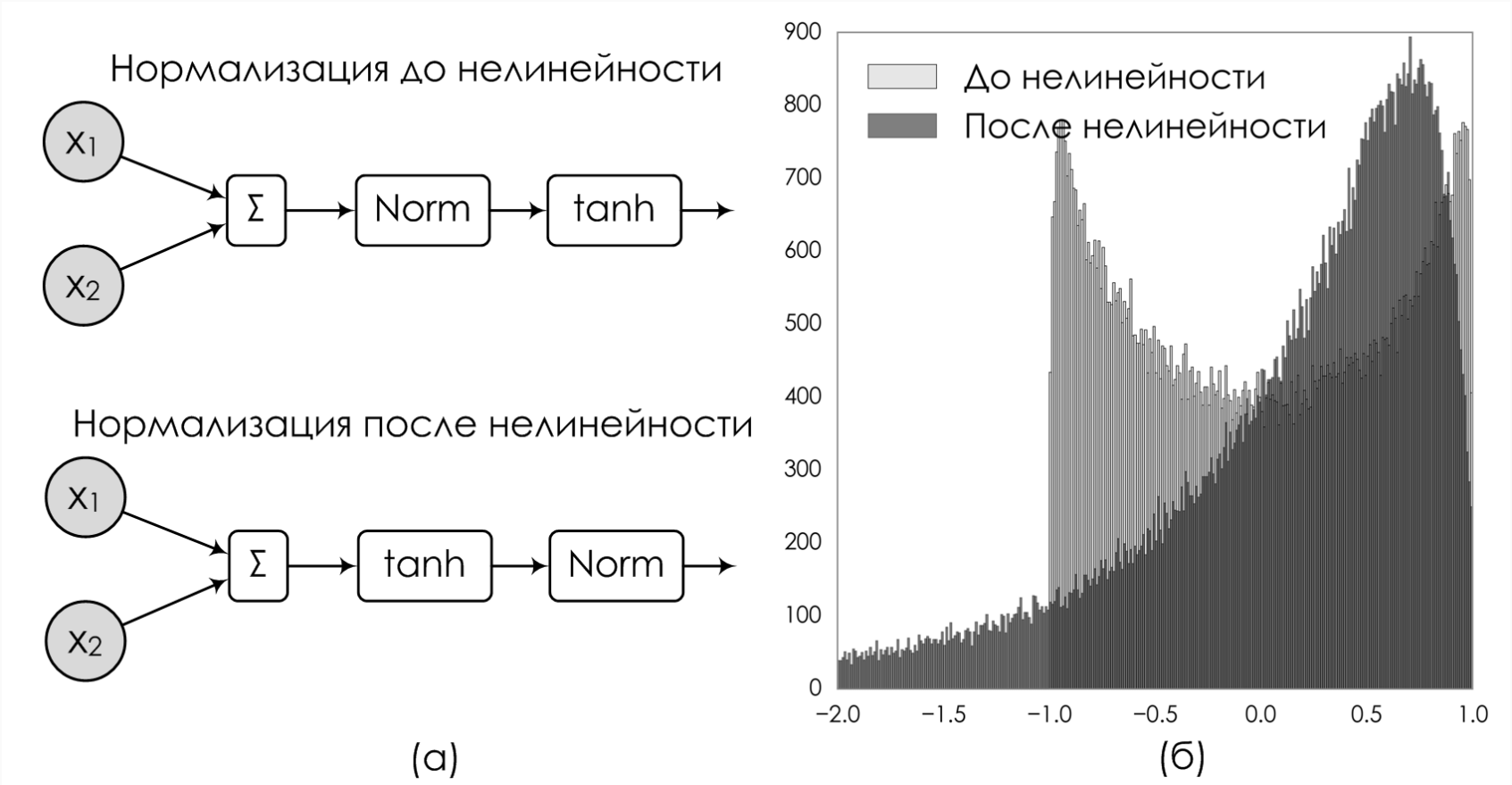
  - вычислить результат

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

- 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.



ВАРИАНТЫ

- 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

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

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

Варианты градиентного спуска

МЕТОД МОМЕНТОВ

- «Ванильный» стохастический градиентный спуск:

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

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

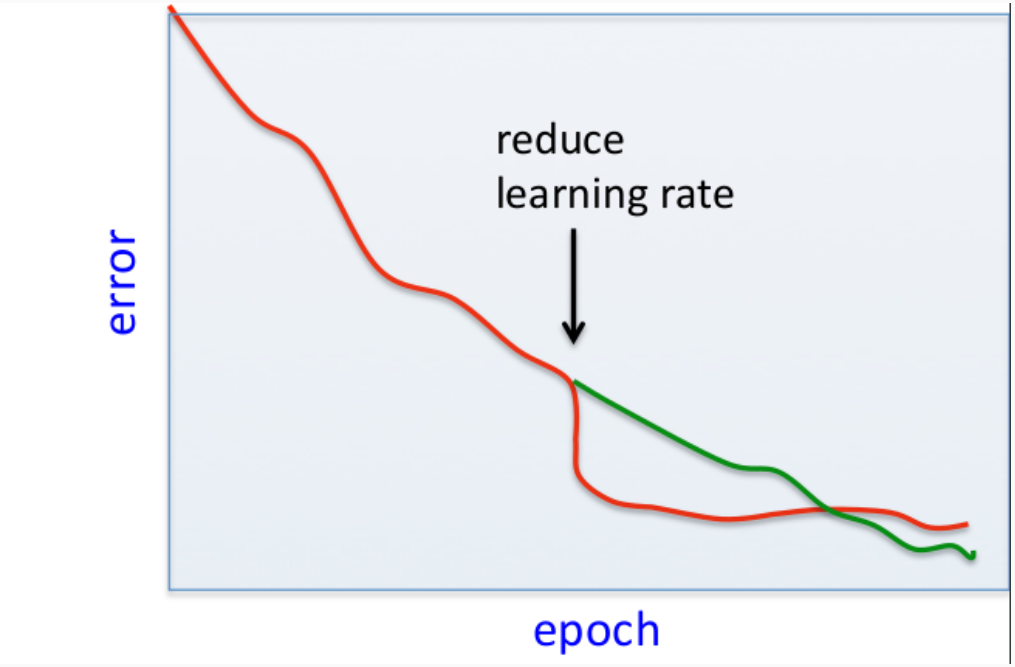
$$\eta = \eta_0 \left(1 - \frac{t}{T}\right);$$

  - или экспоненциально (exponential decay):

$$\eta = \eta_0 e^{-\frac{t}{T}}.$$

МЕТОД МОМЕНТОВ

- Скорость обучения лучше не уменьшать слишком быстро.



- Но это в любом случае никак не учитывает собственно E; лучше быть *адаптивным*.

Метод моментов

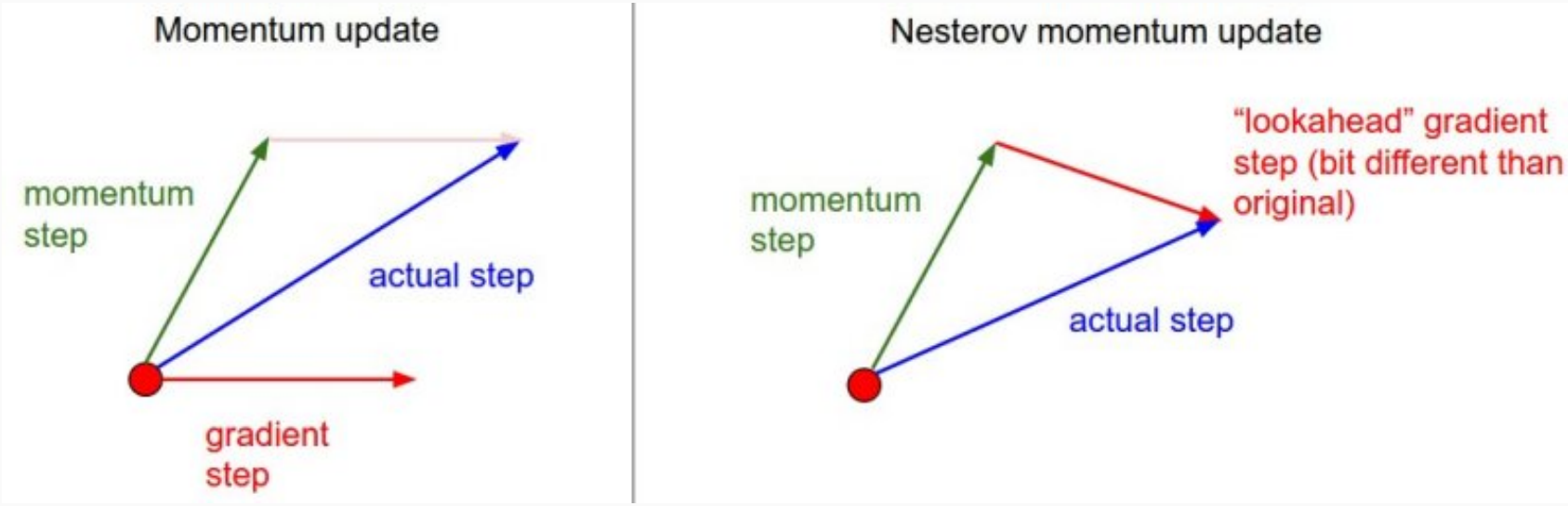
- *Метод моментов* (momentum): сохраним часть скорости, как у материальной точки.
- С инерцией получается

$$u_t = \gamma u_{t-1} + \eta \nabla_\theta E(\theta),$$
$$\theta = \theta - u_t.$$

- И теперь мы сохраняем γu_{t-1}.

МЕТОД МОМЕНТОВ

- Но на самом деле мы 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):

$$u_t = \gamma u_{t-1} + \eta \nabla_\theta E(\theta - \gamma u_{t-1})$$

- Можно ли ещё лучше?..

МЕТОД МОМЕНТОВ

- ...ну, можно попробовать методы второго порядка.
- Метод Ньютона:

$$E(\theta) \approx E(\theta_0) + \nabla_\theta E(\theta_0)(\theta - \theta_0) + \frac{1}{2}(\theta - \theta_0)^\top H(E(\theta))(\theta - \theta_0).$$

- Когда работает, это обычно гораздо быстрее, и нет никакой η, ничего настраивать не надо.
- Но нужно считать гессиан $H(E(\theta))$, и это нереально.

Метод моментов

- Есть, правда, приближения.
- 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 лоб» было бы нужно считать градиент по всему датасету, а по мини-батчам непонятно как).

Адаптивные методы градиентного спуска

- Но дальше улучшить всё равно можно.
- Заметим, что до сих пор скорость обучения была одна во всех направлениях.
- 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 $g_{t,i} = \nabla_{\theta_i} L(\theta)$, получим

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

  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 θ_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:

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

- 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 $\Delta\theta$ — 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 $\Delta\theta$ зависели от отношений градиентов, и величина обновлений вовсе безразмерная.

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

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

- 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:

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

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

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

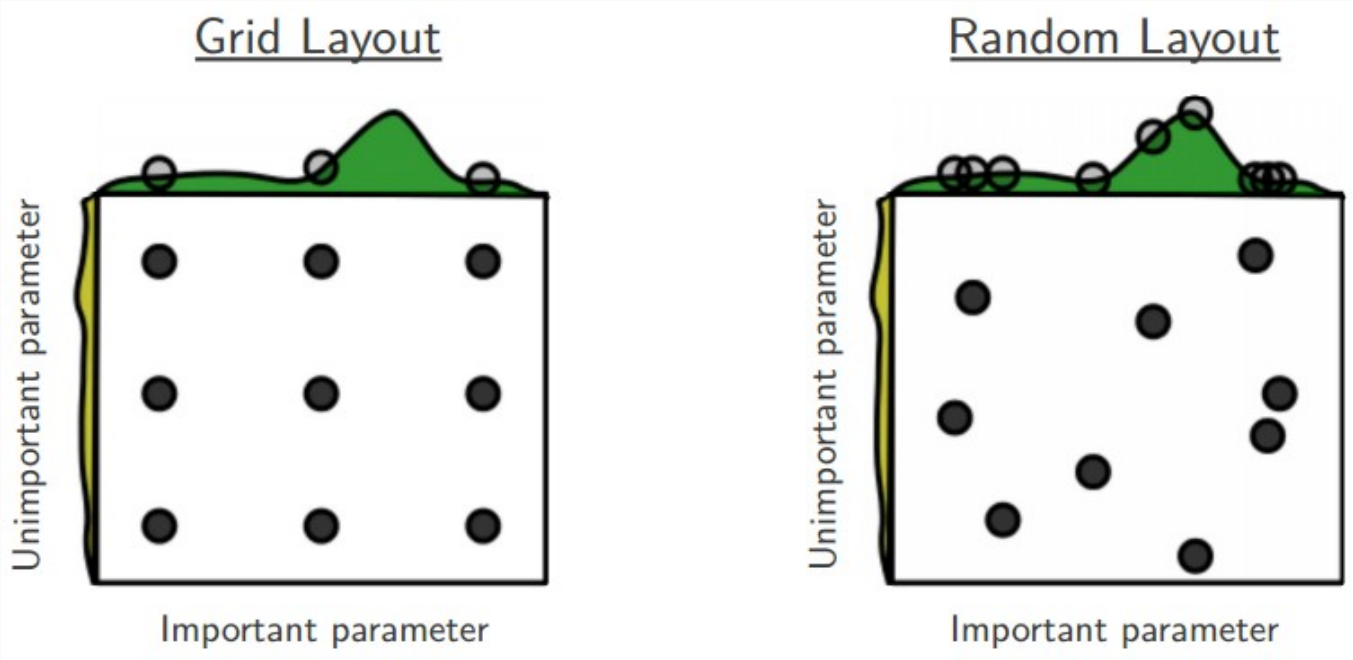
- И последний алгоритм – *Adam* (Kingma, Ba, 2014).
- Модификация *Adagrad* со сглаженными версиями среднего и среднеквадратичного градиентов:

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

- (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.
- ...[анимированные примеры]...

ПРАКТИЧЕСКИЕ ЗАМЕЧАНИЯ

- Ещё практические замечания об оптимизации гиперпараметров:
  - одного валидационного множества достаточно;
  - лучше гиперпараметры искать на логарифмической шкале;
  - и лучше случайным поиском, а не по сетке (Bergstra and Bengio).



СПАСИБО!

Спасибо за внимание!