Лекция 3. Полносвязные нейронные сети

О чём эта лекция

- Что такое полносвязные нейронные сети
- Как они выглядят
- Каковы их возможности
- Как их обучают

Линейные модели

$\sum_{j=1}^{n} x^j w_j + w_0$ -линейная регрессия

$\sigma(\sum_{j=1}^{n} x^j w_j + w_0)$ - логистическая регрессия, $\sigma(z) = \frac{1}{1 + e^{-z}}$

x^j - признаки, w_j - веса признаков. w_0 - bias (может быть любым числом, часто 1 или -1).

Что такое нейрон?

Нейрон - обобщение двух предыдущих моделей:

$$\sigma(\sum_{j=0}^{n} x^j w_j)$$

σ - функция активации нейрона
Когда $\sigma(z) = z$ - получаем линейную регрессию
Когда $\sigma(z) = \frac{1}{1+e^{-z}}$ - получаем логистическую регрессию

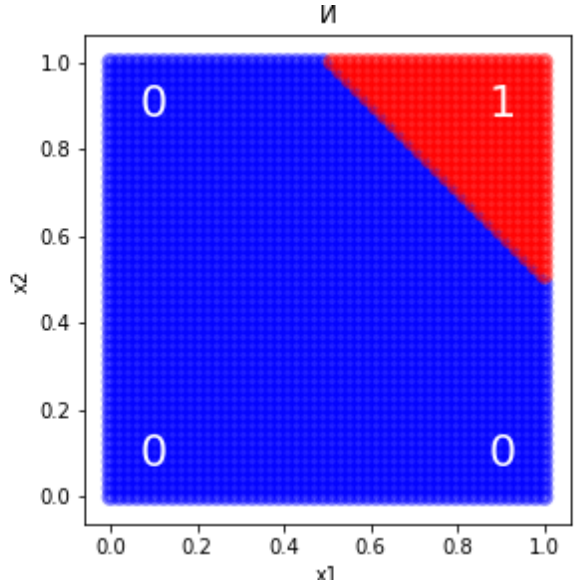
Нейрон = линейная регрессия + функция активации.

Что может один нейрон?

Берем линейную регрессию и функцию акцивации sign. Теперь мы можем:

- Реализовать логическое "И":

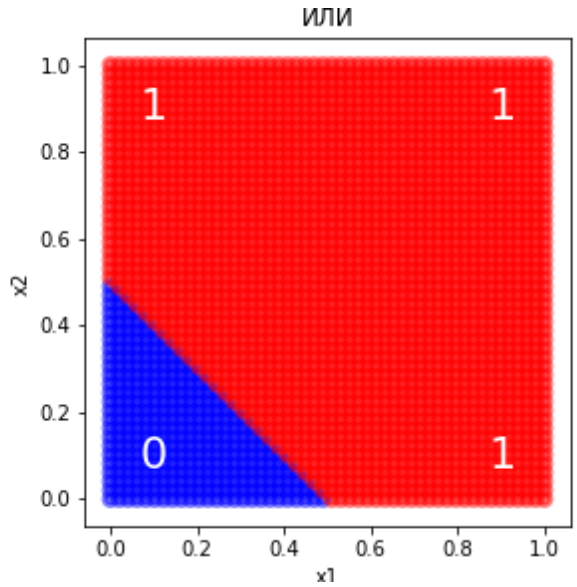
$$x^1 \wedge x^2 = x^1 + x^2 - 1.5 > 0$$



Что может один нейрон?

- Реализовать логическое "ИЛИ":

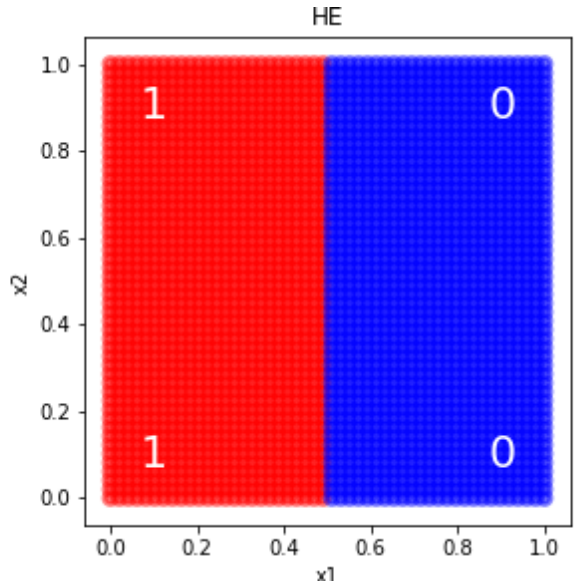
$$x^1 \vee x^2 = x^1 + x^2 - 0.5 > 0$$



Что может один нейрон?

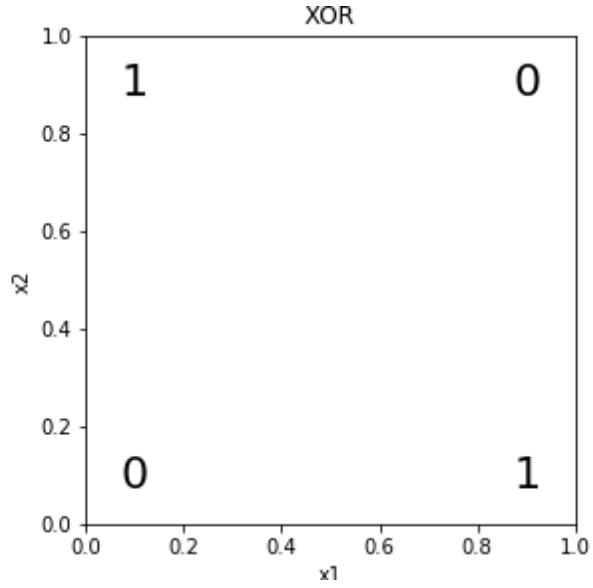
- Реализовать логическое "НЕ":

$$\neg x^1 = -x^1 + 0.5 > 0$$



Что не может один нейрон?

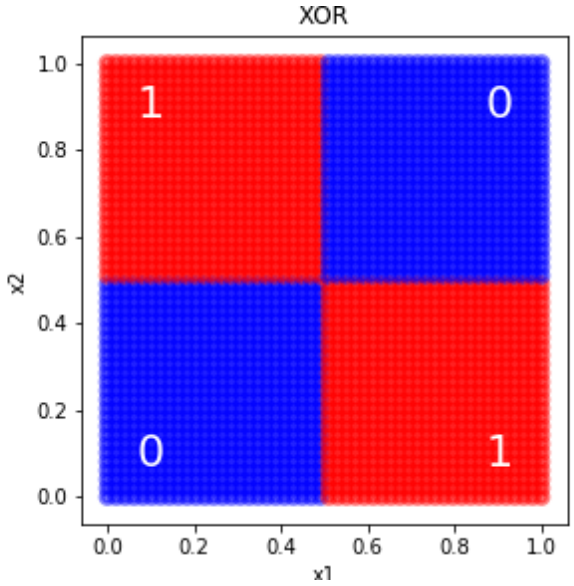
- Линейная модель строит линейную раздедяющую поверхность.
- Как тогда решить такую задачу?



Задача XOR

- Первый способ: добавить ещё один признак $x^3 = x^1 x^2$
  - Тогда в пространстве из трёх признаков можно построить линейную разделяющую поверхность

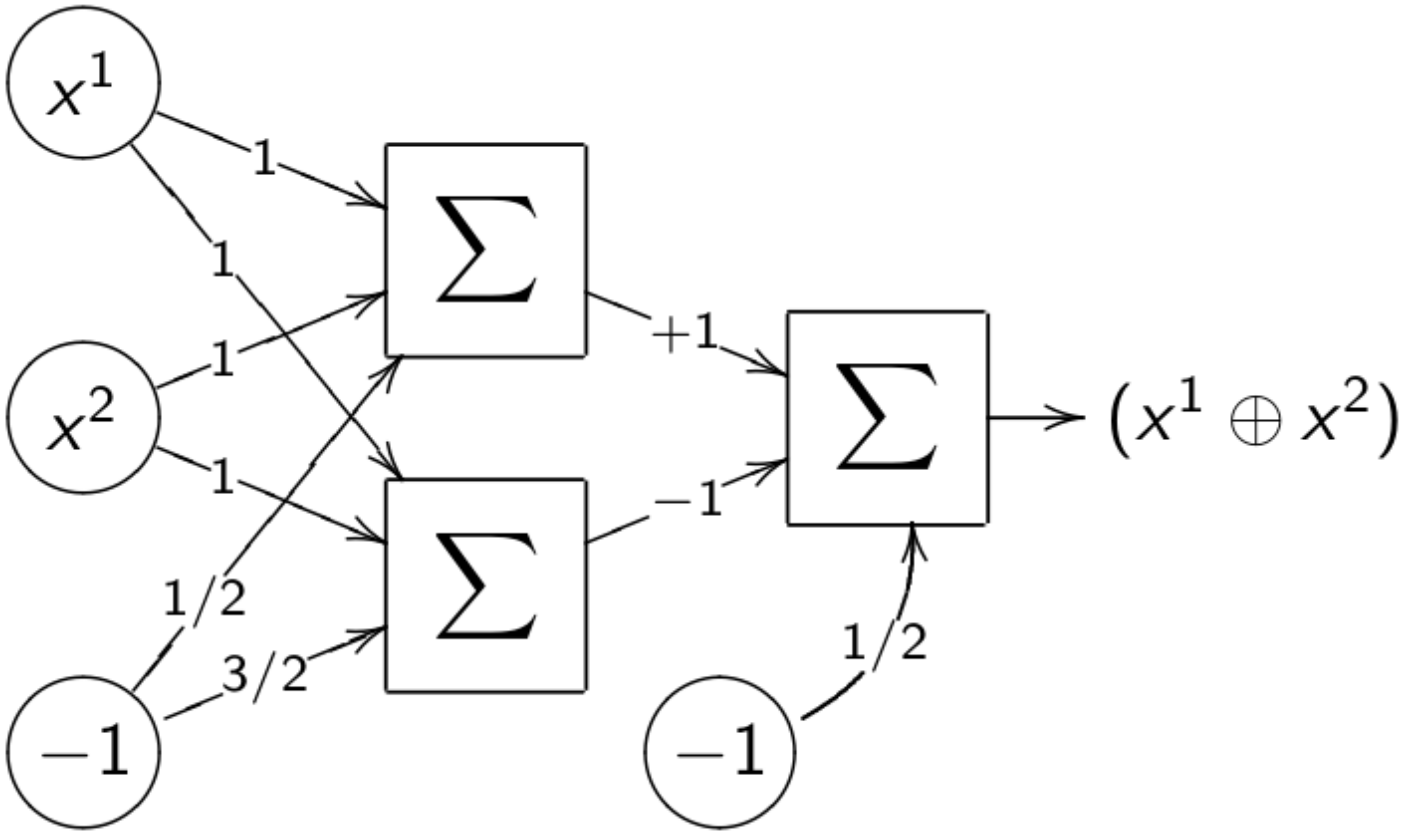
$$x^1 \oplus x^2 = x^1 + x^2 - 2x^1 x^2 - 0.5 > 0$$



Задача XOR

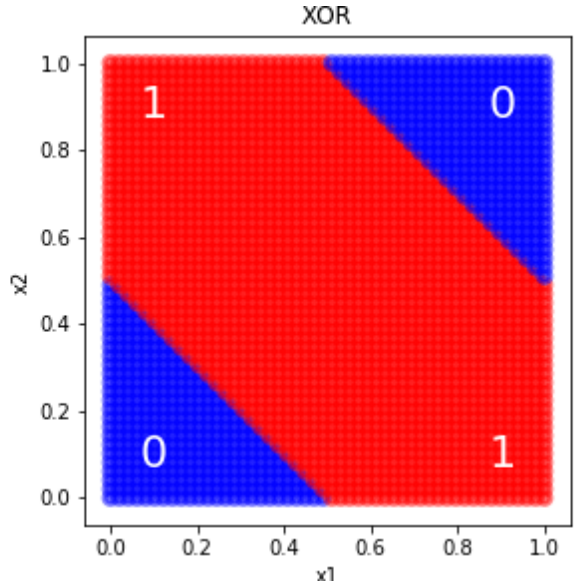
- Второй способ: построить нейронную сеть, использовать выходы одних нейронов как входы для других
- (соединение линейных регрессий с функциями активаций)

$$x^1 \oplus x^2 = (x^1 \vee x^2) - (x^1 \wedge x^2) - 0.5 > 0$$



Задача XOR

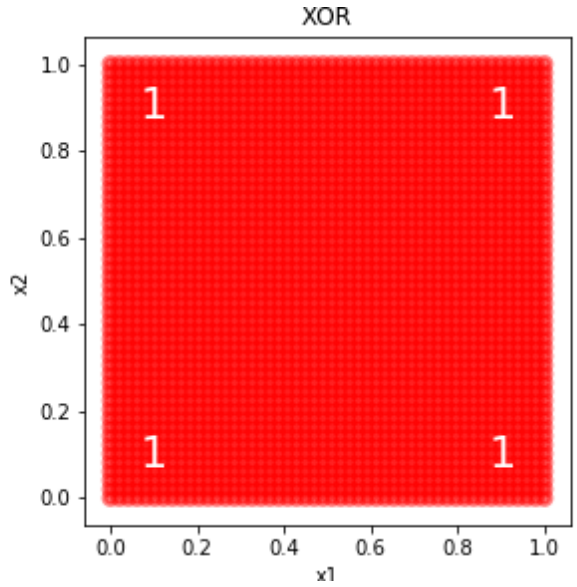
Нейронная сеть с предыдущего слайда строит нелинейную разделяющую поверхность:



Задача XOR

Важно - мы использовали *нелинейную* функцию активации (sign) внутри сети.

- Вот как выглядит результат, если мы уберем sign везде кроме выхода (просто используем композицию линейных регрессий)
- В данном случае мы даже не получим разделяющей поверхности
- В общем случае модель останется линейной и мы получим лишь линейную разделяющую поверхность



Нейронные сети как граф вычислений

Известная из мат. логики формула

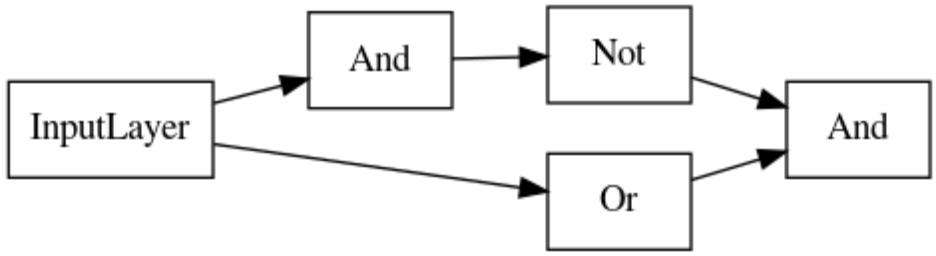
$$x^1 \oplus x^2 = (x^1 \vee x^2) \wedge \neg(x^1 \wedge x^2)$$

где $\wedge, \vee, \neg$ - нейроны с предыдущих слайдов и функцией активации $sign$.

Результат - нейронная сеть, которую мы рассмотрели на предыдущих слайдах.

- Нейронная сеть - граф вычислений (суперпозиция слоев - функций) над признаками.
- Таким образом с ними работают в современных библиотеках (tensorflow, pytorch, keras...)
- Это язык описания архитектур нейронных сетей

Какие бывают архитектуры нейронных сетей?



Виды нейронных сетей

Существуют разные типы слоев (операций) и типы архитектур нейронных сетей:

- полносвязные
- сверточные
- рекуррентные
- рекурсивные

Виды нейронных сетей

При этом:

- В сверточных сетях используются полносвязные слои
- В рекуррентных сетях аналогично могут использоваться свертки
- и так далее

Почему?

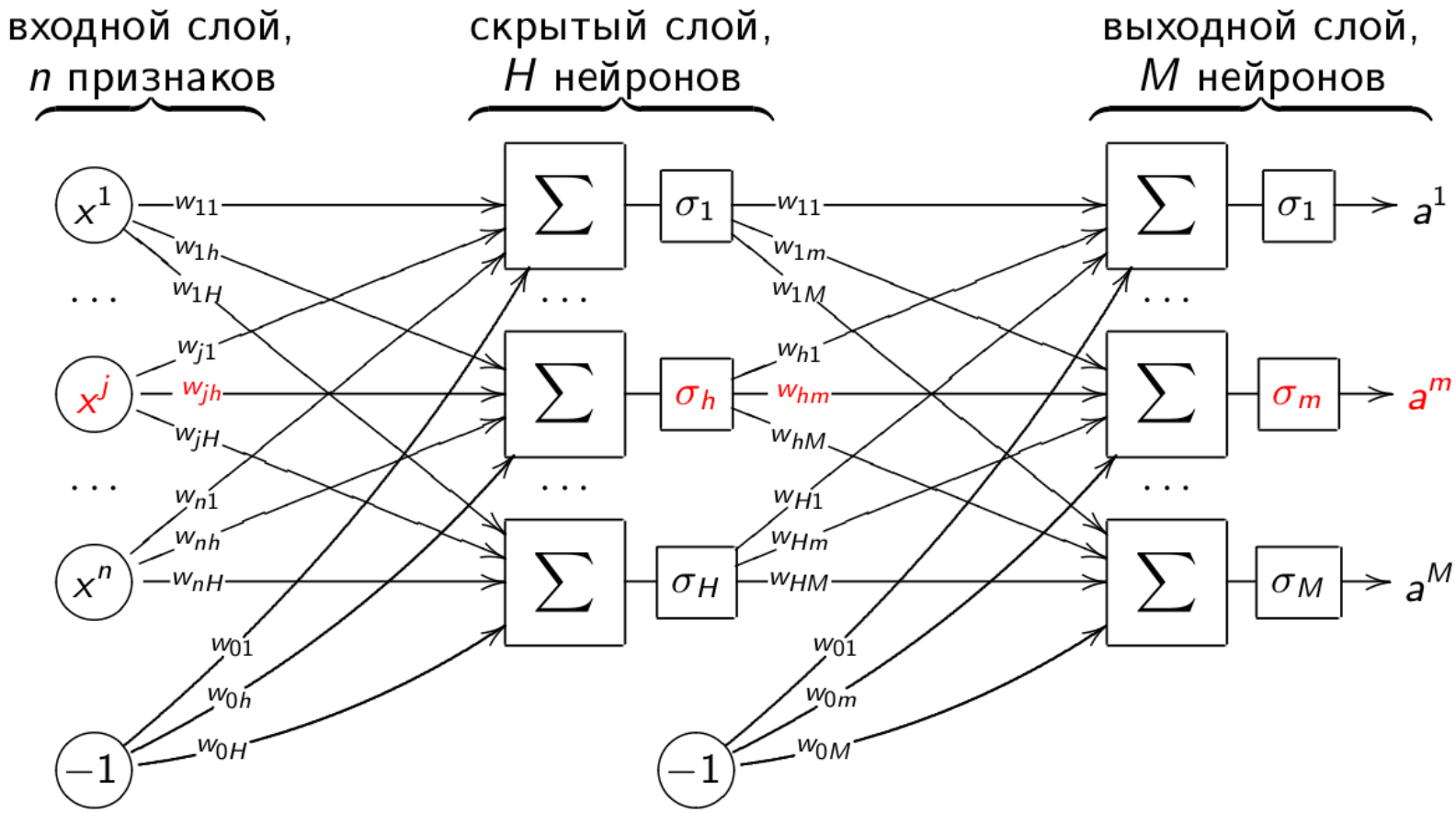
- Так происходит т.к. сеть - граф вычислений в котором могут встречаться разные операции

Далее рассмотрим самый простой тип - полносвязные нейронные сети.

Пример полносвязной нейронной сети

У полносвязной нейронной сети каждый нейрон текущего слоя связан с каждым нейроном предыдущего.

Пример для двух слоев:



Нейронные сети

Вопрос: Чем параметры модели отличаются от гиперпараметров?

Нейронные сети - параметры и гиперпараметры

Обучаемые параметры:

- Веса сети - w_{ij} - вес связи нейрона\признака i и нейрона j

Гиперпараметры:

- Архитектура сети
    - Количество слоев
    - Количество нейронов в слоях
    - Используемые функции активации
    - ...

На практике встречаются исключения, например функции активации с обучаемыми параметрами

Нейронные сети - реальность

Что говорим:

- Нейроны, связи, веса, активации...

Что происходит *на самом деле*:

- Матрицы умножаются на вектора и от них вычисляются нелинейные функции

$$\sigma(\sum w_{ji}x_{kj}) = \sigma(W^T x)$$

Это причина почему нейронные сети обучают на GPU:

- архитектура SIMD (single instruction multiple data)
- можно быстро умножать матрицы и применять функции на векторы

Выразительная способность нейронных сетей

- Нейронная сеть с одним скрытым слоем может отделить в R^n любой многогранник
- Нейронная сеть с двумя скрытыми слоями может отделить в R^n произвольную многогранную область
    - возможно не связную
    - возможно не выпуклую
- Нейронной сетью с одним скрытым слоем и нелинейной функцией активации (предыдущий слайд) можно приблизить любую непрерывную функцию с наперед заданной точностью

Выразительная способность нейронных сетей

Таким образом нейронные сети являются универсальными аппроксиматорами.

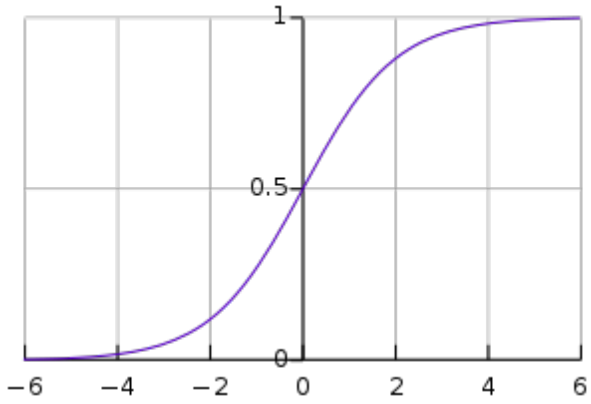
- Теоретически двух-трёх слоев достаточно
- На практике часто используют глубокие нейронные сети для встроенного обучения признаков

Функции активации

- На практике применяют разные функции активации
- Обычно стараются применять дифференциируемые почти везде (кроме конечного количества точек)

Очень часто используются:

- $\sigma(x) = \frac{1}{1 + e^{-x}}$ сигмоида, например, на выходе для задачи бинарной классификации

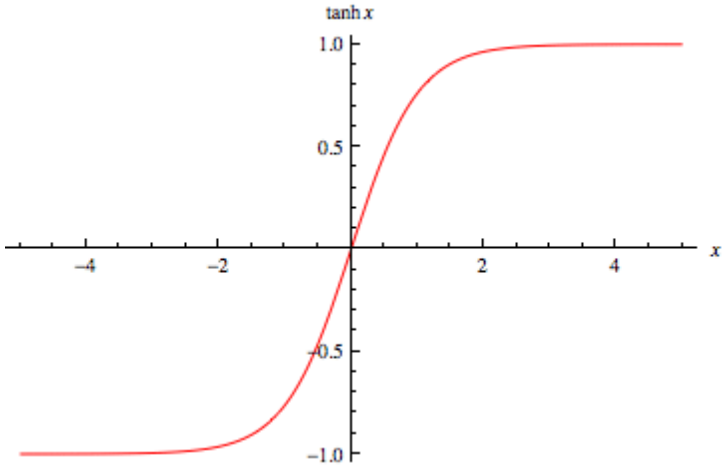


- $softmax(x_j) = \frac{e^{x_j}}{\sum_i e^{x_i}}$ - софтмакс, чаще всего на выходе для задачи с несколькими непересекающимися классами

Функции активации

$tanh(x)$ - гиперболический тангенс, предлагался как альтернатива $\sigma(x)$ на скрытых слоях сети

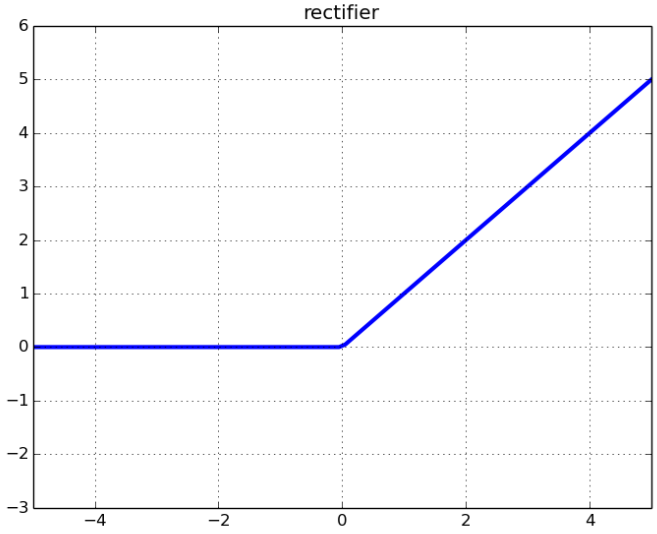
- В отличии от $\sigma(x)$ имеем $tanh(0) = 0$
- Сети с использованием этой активации легче оптимизируются в сравнении с теми что используют сигмоиду



Функции активации

$ReLU(x) = max(x, 0)$ - REctified Linear Unit, широко используется сейчас для скрытых слоёв сетей

- Хорошо показала себя на практике
  - Сети обычно быстрее сходятся в сравнении с sigmoid/tanh
- Легче вычислить в отличие от $\sigma(x)$ и $tanh(x)$
- Активации достаточно часто могут быть равны нулю
  - Есть модификации, например, LeakyReLU - когда значение !=0 когда x<0



Обучение нейронных сетей

- Как-правило для обучения сети требуется довольно большая выборка
- Функция потерь нейронной сети на практике не выпуклая

Поэтому для обучения нейронных сетей сейчас применяют вариации метода стохастического градиентного спуска (как правило с использованием mini-batch).

Обучение нейронных сетей

Проблема:

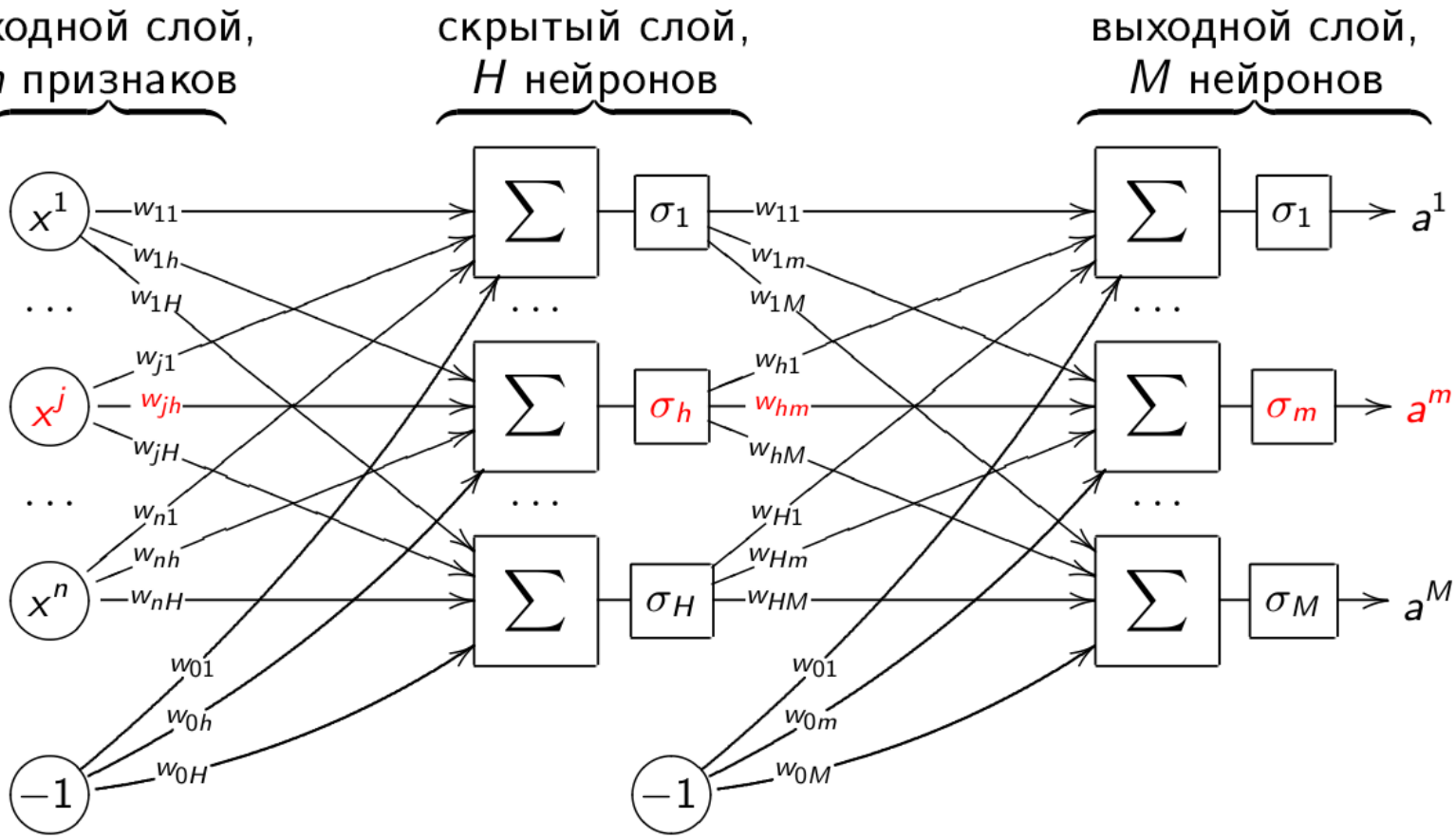
На каждом шаге стохастического градиентного спуска нужно вычислить градиент ошибки. Как это сделать эффективно?

Решение:

Метод обратного распространения ошибки (Backprop)

Алгоритм обратного распространения ошибки

Пример:



Алгоритм обратного распространения ошибки

Пример:

- Скрытый слой

$$u_h(x) = \sigma_h(\sum_{j=0}^{n} w_{jh} x^j)$$

Здесь w_{jh} - вес связи признака x^j примера x и нейрона h скрытого слоя

- Выходной слой

$$a_m(x) = \sigma_m(\sum_{h=0}^{H} w_{hm} u_h(x))$$

Здесь w_{hm} - вес связи нейрона h скрытого слоя и нейрона m выходного слоя

- Среднеквадратичная функция потерь

$$SE(x, y) = \frac{1}{2} \sum_{m=1}^{M} (a_m(x) - y_m)^2$$

Алгоритм обратного распространения ошибки

Задача:

- Найти градиент - вектор производных по параметрам модели

Вопрос залу:

- Что за параметры?

Алгоритм обратного распространения ошибки

Нужно найти $\nabla SE(w) = (\frac{\partial SE(w)}{dw_{ij}})$ как функции от весов сети.

$$\frac{\partial SE}{dw_{hm}} = ???$$

$$\frac{\partial SE}{dw_{jh}} = ???$$

Алгоритм обратного распространения ошибки

Решение: применить правило дифф. сложной функции:

$$\frac{\partial SE}{dw_{hm}} = \frac{\partial SE}{da_m}\frac{\partial a_m}{dw_{hm}}$$

$$\frac{\partial SE}{dw_{jh}} = \frac{\partial SE}{du_h}\frac{\partial u_h}{dw_{jh}}$$

Алгоритм обратного распространения ошибки

Производная по весам выходного слоя:

$$\frac{\partial SE}{dw_{hm}} = \frac{\partial SE}{da_m}\frac{\partial a_m}{dw_{hm}}$$

$$\frac{\partial a_m}{dw_{hm}} = \frac{\partial \sigma_m(\sum_{h=0}^{H} w_{hm}u_h(x))}{dw_{hm}} =$$

Правило дифф. сложной функции:

$$= \sigma'_m u_h(x)$$

Получаем:

$$\frac{\partial SE}{dw_{hm}} = \frac{\partial SE}{da_m}\frac{\partial a_m}{dw_{hm}} = \frac{\partial SE}{da_m}\sigma'_m u_h(x)$$

Алгоритм обратного распространения ошибки

$$\frac{\partial SE(w)}{da_m} = \frac{\partial \frac{1}{2} \sum_{m=1}^{M} (a_m(w) - y_m)^2}{da_m} = a_m(w) - y_m = \epsilon_m$$

ϵ_m - значение производной выше, ошибка модели на выходном слое.

Теперь можно посчитать ошибку по весам выходного слоя:

$$\frac{\partial SE}{dw_{hm}} = \frac{\partial SE}{da_m} \frac{\partial a_m}{dw_{hm}} = \frac{\partial SE}{da_m} \sigma'_m u_h(x) = \epsilon_m \sigma'_m u_h(x)$$

Алгоритм обратного распространения ошибки

Посмотрим на ошибку модели на скрытом слое:

$$\frac{\partial SE}{dw_{jh}} = \frac{\partial SE}{du_h} \frac{\partial u_h}{dw_{jh}}$$

- Правило дифф. сложной функции:

$$\frac{\partial SE(w)}{du_h} = \sum_{m=1}^{M} \frac{\partial SE}{da_m} \frac{\partial a_m}{du_h} =$$

- Подстановка известного значения производной по a_m:

$$= \sum_{m=1}^{M} \epsilon_m \frac{\partial a_m}{du_h} =$$

- По определению $a_m(x)$:

$$= \sum_{m=1}^{M} \epsilon_m \frac{\partial \sigma_m(\sum_{h=0}^{H} w_{hm} u_h(x))}{du_h} =$$

Вопрос:

Что делаем дальше?

Алгоритм обратного распространения ошибки

... продолжаем искать производную ошибки на скрытом слое:

- Правило дифф. сложной функции, второй раз:

$$= \sum_{m=1}^{M} \epsilon_m \sigma'_m \frac{\partial \sum_{h=0}^{H} w_{hm} u_h(x)}{du_h} =$$

- Результат:

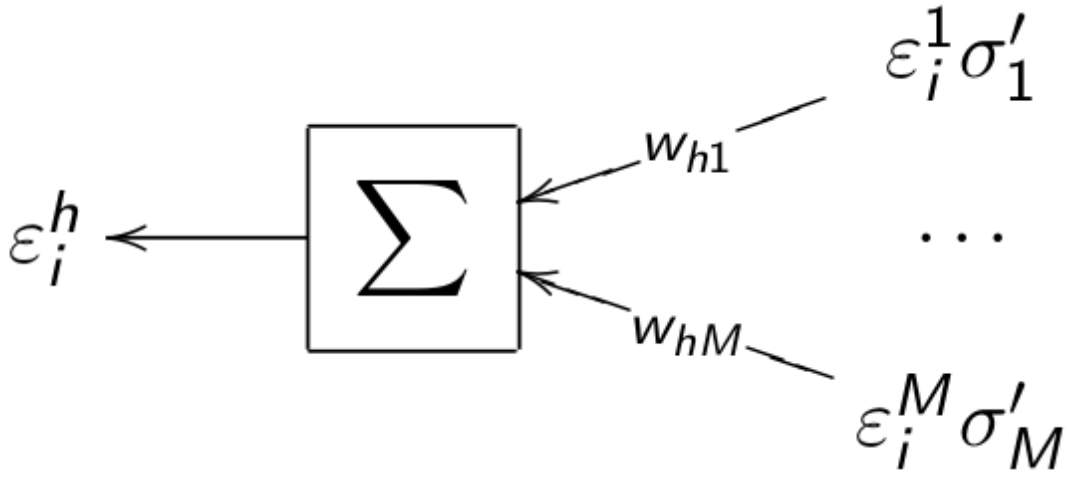
$$= \sum_{m=1}^{M} \epsilon_m \sigma'_m w_{hm}$$

Алгоритм обратного распространения ошибки

- Теперь можно посчитать производные по весам скрытого слоя:

$$\frac{\partial SE(w)}{du_h} = \sum_{m=1}^{M} \epsilon_m \sigma'_m w_{hm} = \epsilon_h$$

$$\frac{\partial SE}{dw_{jh}} = \frac{\partial SE}{du_h}\frac{\partial u_h}{dw_{jh}} = \epsilon_h \frac{\partial u_h}{dw_{jh}} = \epsilon_h \frac{\partial u_h}{dw_{jh}} = \epsilon_h \sigma'_h x^j$$



Алгоритм обратного распространения ошибки

Наблюдения:

- значение производной можно получить запустив сеть "наоборот"
- вычисление производной суть последовательное применение правила дифф. сложной функции над компонентами сети:
  - слоями
  - функциями активаций

Окончательное решение:

$$\frac{\partial SE}{dw_{hm}} = \frac{\partial SE}{da_m}\frac{\partial a_m}{dw_{hm}} = \epsilon_m \sigma'_m u_h(x)$$

$$\frac{\partial SE}{dw_{jh}} = \frac{\partial SE}{du_h}\frac{\partial u_h}{dw_{jh}} = \epsilon_h \sigma'_h x^j$$

Алгоритм обратного распространения ошибки

- Прямой ход:
    - Применить слой за слоем на данный пример, посчитать значение функции потерь от выходов
- Обратный ход:
    - Считать значения производных слой за слоем в обратном направлении от выходов к входам
        - Используя правило дифференцирования сложной функции
        - Считать сначала производные ошибки от выходов, затем производные ошибки от последнего скрытого слоя и т.д. пока не будут расчитаны производные по всем весам

Посчитаный градиент использовать для расчета шага в стохастическом градиентном спуске.

Алгоритм обратного распространения ошибки на графе

Рассмотрим общий случай:

- нейронная сеть - это граф вычислений
- Вершины - операции:
    - Умножение вектора на матрицу
    - Применение функции активации

Вопрос:
Что нужно уметь считать для каждой вершины графа, чтобы реализовать алгоритм обратного распространения ошибки?

Алгоритм обратного распространения ошибки на графе

Алгоритм backprop применяют к нейронным сетям как к графу вычислений. Для каждой вершины нужно знать:

- как по входам вычислить её значение (прямой ход)
- как посчитать её производная от выходов (обратный ход)

Алгоритм обратного распространения ошибки

Таким образом построение нейронных сетей превращается из упражнений по дифференцированию и матричной алгебре в описание графов вычислений из заданных компонент (вершин). Обычно предоставляются:

- автоматическое дифференцирование графов - алгоритм обратного распространения ошибки
- набор компонент для построения сетей (слои, функции активации...)
- набор оптимизаторов для обучения

Далее поговорим подробнее об обучении и борьбе с переобучением.

Momentum

Существует приличное множество модификаций стохастического градиентного спуска, все из которых используются на практике при обучении сетей:

- Обычный градиентный шаг $\eta_t \nabla_t$
- Momentum - экспоненциальное скользящее среднее по $\approx \frac{1}{1-\gamma}$ итерациям спуска:

$$h_t = \alpha h_{t-1} + \eta_t \nabla_t$$
$$w_t = w_{t-1} - h_t$$

  - На деле это напоминает среднее по N последним итерациям спуска
  - Старается двигаться в том-же направлении что и на предыдущих итерациях
  - h_t растет в направлении где градиенты с разных шагов чаще положительны
  - Обычно $\alpha = 0.9$

Nesterov Momentum

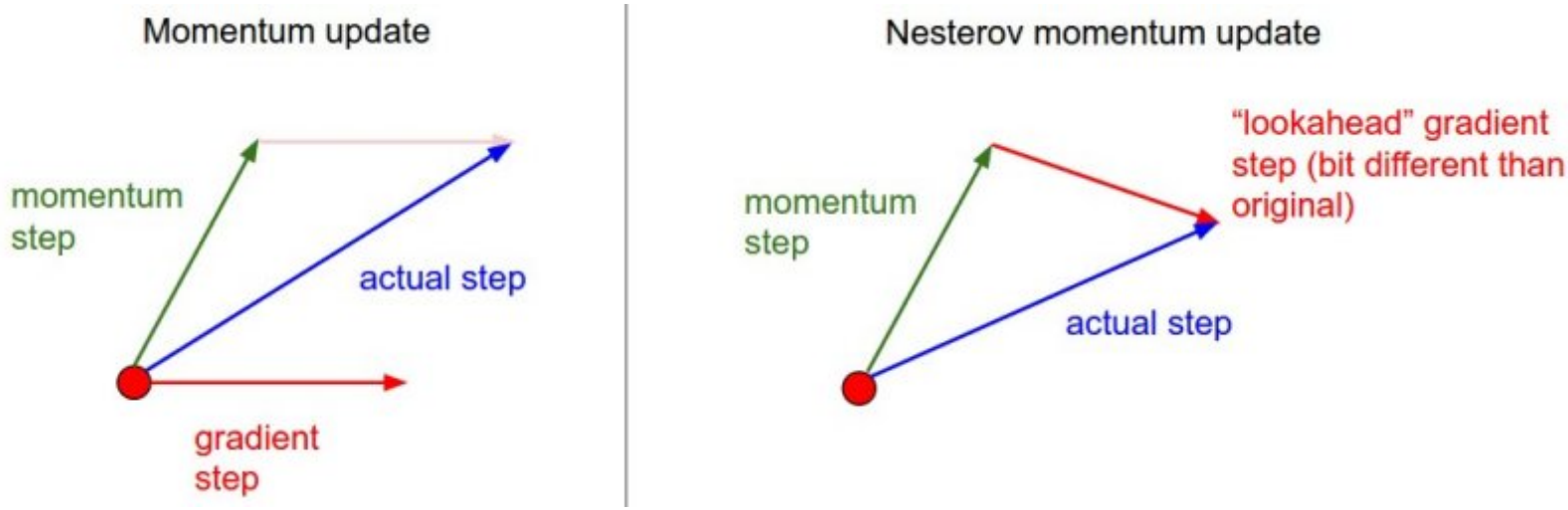
- Nesterov momentum:

$$h_t = \alpha h_{t-1} + \eta_t \nabla L(w_{t-1} - \alpha h_{t-1})$$
$$w_t = w_{t-1} - h_t$$

- Тоже что и momentum, но градиентный шаг идет из предполагаемой точки где мы окажемся, а не из текущей

Momentum

- Физическая аналогия для этих эвристик - накопление импульса при спуске (шар с ненулевой массой)



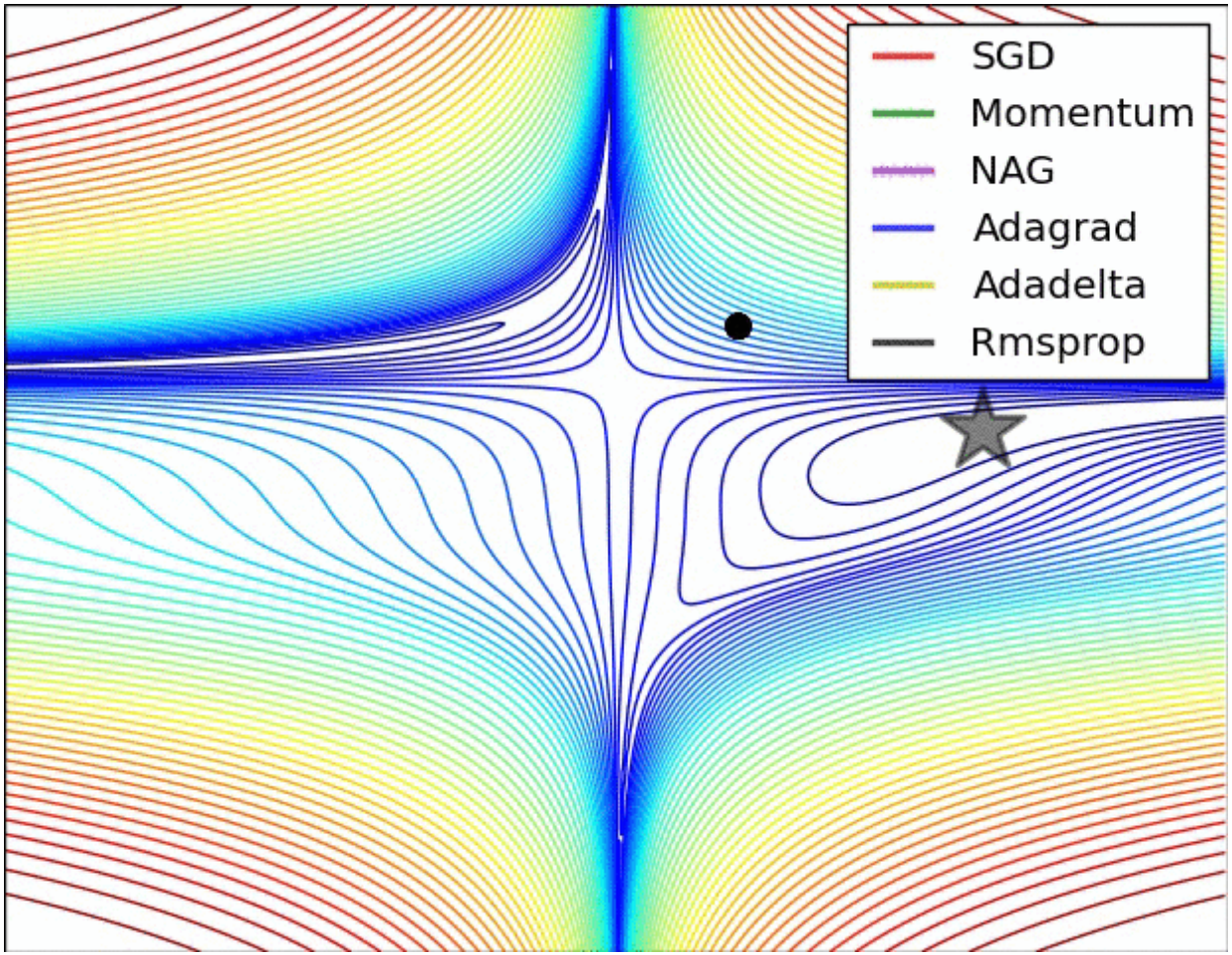
Методы с адаптивной скоростью обучения

- Adagrad:
    - Скорость обучения своя под каждый параметр:
        - Большая производная от параметра - скорость обучения падает
        - Маленькая производная - скорость обучения растет
    - Подходит для разреженных данных
    - Имеет тенденцию к ранней остановке (до достижения локального минимума)
- RMSProp:
    - decay_rate - гиперпараметр с типичными значениями 0.9, 0.99, 0.999
    - В отличие от Adagrad обновления скорости обучения зависят от последних шагов - нет проблемы ранней остановки

Методы с адаптивной скоростью обучения

- Adam - сейчас чаще всего используется на практике
  - Объединяет RMSProp и momentum
  - Рекомендуемые значения гиперпараметров `eps = 1e-8`, `beta1 = 0.9`, `beta2 = 0.999`

Визуализация сходимости методов



Визуализация сходимости методов

Наблюдение: Несмотря на то что это всего-лишь эвристики, они обладают большим влиянием на скорость сходимости спуска.

Dropout

Идея:

- Во время обучения (но не применения!) будем случайным образом "выключать" нейроны

Мотивы:

- получаем приближение ансамбля из 2^N сетей с общими весами, но разными связями между нейронами
- тренируем сеть наиболее устойчивую к утрате нейронов надеясь что она будет надежной
- заставляем разные части сети решать одну и ту-же задачу, а не компенсировать ошибки других частей

Результат:

Отличный метод борбы с переобучением нейронных сетей

Инициализация весов

Почему важно правильно инициализировать веса?

Стохастический градиентный спуск - ищет минимум в окрестности некоторой точки.

Правильная инициализация весов дает возможность найти лучший локальный минимум.

Инициализация весов

Как **не надо** делать:

- инициализация одной константой
    - В этом случае все веса так и останутся одинаковыми во время обучения (т.к. получат одинаковые градиенты)

Для неглубоких сетей (2-5 слоев):

- инициализация небольшими случайными значениями около нуля

Инициализация весов в глубоких сетях

Для глубоких сетей применяют спец. методы исходящие из статистических соображений.

- Для симметричных функций активации с нулевым средним (например tanh):
  - инициализация Ксавье

$$w_i \sim \mathcal{N}(0, \sqrt{\frac{2}{N_{in} + N_{out}}})$$

    где N_{in}, N_{out} - размеры входа и выхода слоя

- Для остальных (ReLU, sigmoid...):
  - Инициализация Хе

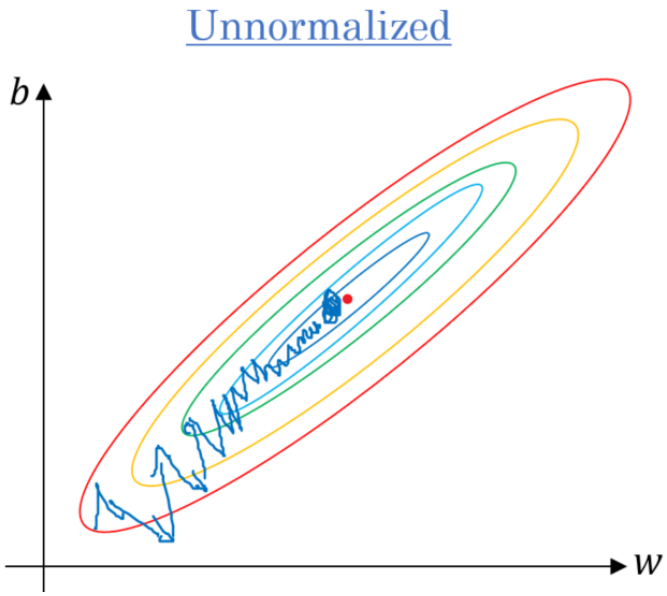
$$w_i \sim \mathcal{N}(0, \sqrt{\frac{2}{N_{in}}})$$

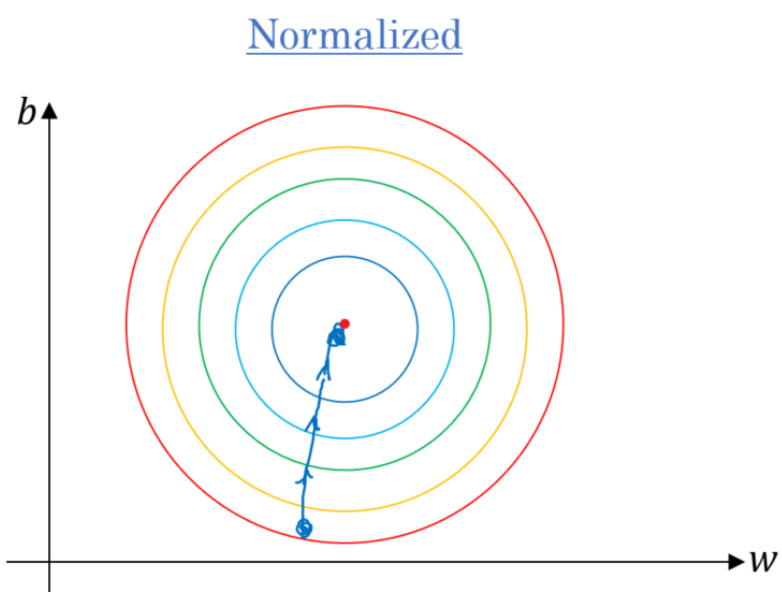
- Часто вместо нормального берут равномерное распределение.

Нормализация

Почему полезно?

- Ускоряет сходимость градиентного спуска
- Аналогия: что если сделать тоже самое не для входных данных, а внутри сети?





Нормализация по мини-батчам

Идея: нормализовать выходы слоев нейронной сети

$$x_{norm} = \frac{x - \mathbb{E}[x]}{Var(x)}$$

- упростит обучение следующего слоя (значения входов не будут сильно сдвигаться по модулю во время обучения)
- решит проблему "насыщения" функций $tanh, sigmoid$ - когда большие по модулю значения дают маленькое изменение после активации

Проблемы:

- Нормализация по всей обучающей выборке? Неприемлимо трудоемко!
- Как учесть нормализацию в алгоритме обратного распространения ошибки?

Нормализация по мини-батчам

Проблемы:

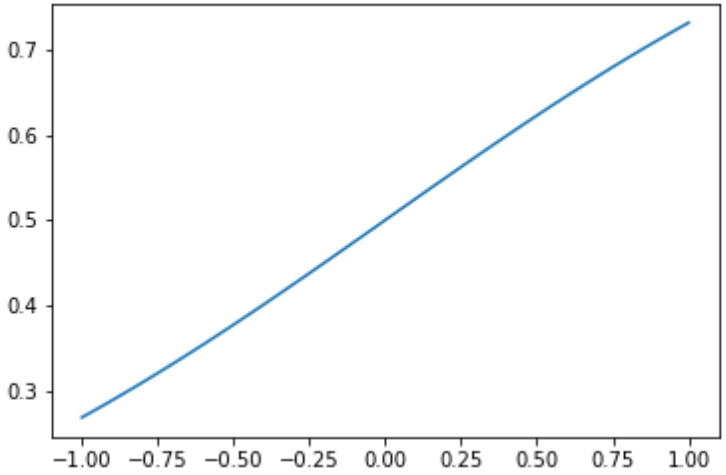
- Нормализация по всей обучающей выборке?
- Как учесть нормализацию в алгоритме обратного распространения ошибки?

Решение:

- Нормализуем не выход целиком, а каждую компоненту отдельно
- Считаем выборочное средние и стандартное отклонение не на всей выборке, а на текущем мини-батче
    - Теперь вычисление локально только для текущего мини-батча
    - Вычисленные значения - приближения таковых для всей выборки
    - Можно посчитать производные для всех весов учавствующих в нормализации т.е. шаг градиентного спуска будет учитывать нормализацию

Нормализация по мини-батчам

Проблема: график сигмоиды в отрезке [-1, 1]:



- Наивная нормализация нейтрализует нелинейности функций sigmoid, tanh

Нормализация по мини-батчам

Решение:

- Добавить два обучаемых параметра: γ - масштаб, β - сдвиг.
- В случае необходимости сеть может сама подвинуть нормализацию и восстановить нелинейность

Таким образом итоговая формула (для мини-батча размером m).

Выборочное среднее по мини-батчу:

$$\mu_{\mathcal{B}} = \frac{1}{m}\sum_{i=1}^{m} x_i$$

Выборочная дисперсия по мини-батчу:

$$\sigma^2_{\mathcal{B}} = \frac{1}{m}\sum_{i=1}^{m}(x_i - \mu_{\mathcal{B}})^2$$

Нормализованный выход слоя:

$$\hat{x_i} = \frac{x_i - \mu_{\mathcal{B}}}{\sqrt{\sigma^2_{\mathcal{B}} + \epsilon}}$$

Сдвинутый\масштабированный нормализованный выход слоя

$$y_i = \gamma\hat{x_i} + \beta$$

Нормализация по мини-батчам

До или после активации?

- Авторы метода рекомендовали применять её до нелинейности (функции активации)
- На практике часто оказывается что модель с BatchNorm после активации работает лучше

Нормализация по мини-батчам

Эффекты:

- Сильное ускорение сходимости глубоких сетей
- Регуляризация, часто применяется в глубоких сетях вместо dropout слоев

Заключение

- Нейронные сети - универсальный аппроксиматор
- На практике сейчас их описывают на языке графов вычислений
- Нейронные сети обычно обучаются вариациями стохастического градиентного спуска
    - градиент вычисляют методом обратного распространения ошибки
- Существует очень большое количество эвристик связанных с обучением нейронных сетей
    - Много модификаций алгоритма стохастического градиентного спуска
    - Дополнительные методы регуляризации (dropout)
    - Специальные методы инициализации весов
    - Нормализация по мини-батчам