Введение в нейронные сети

Лектор — Латыпова Екатерина

Повторение

Что такое нейронные сети?

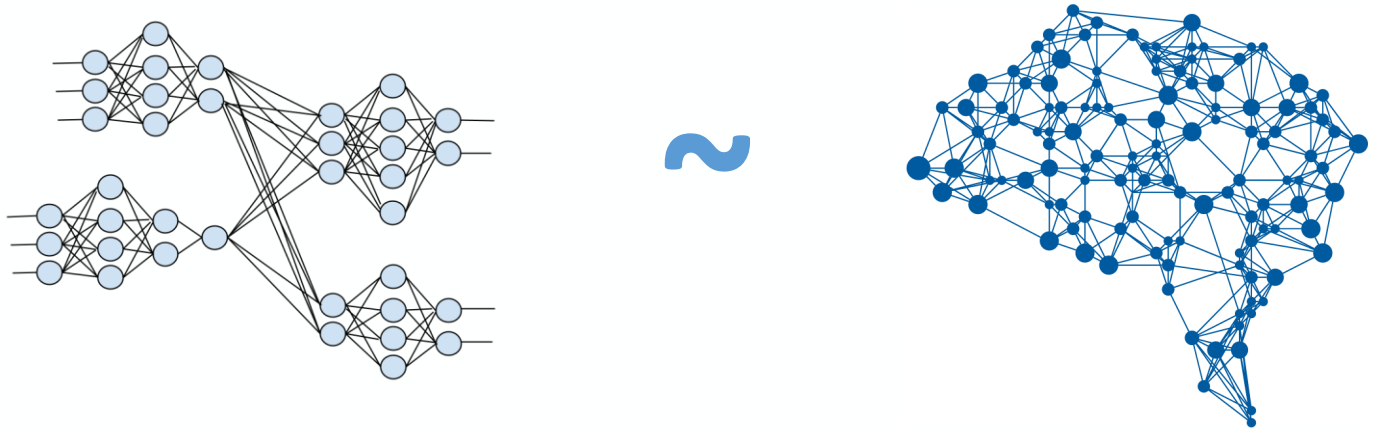
Нейронная сеть — математическая модель,

а также ее программное или аппаратное воплощение,

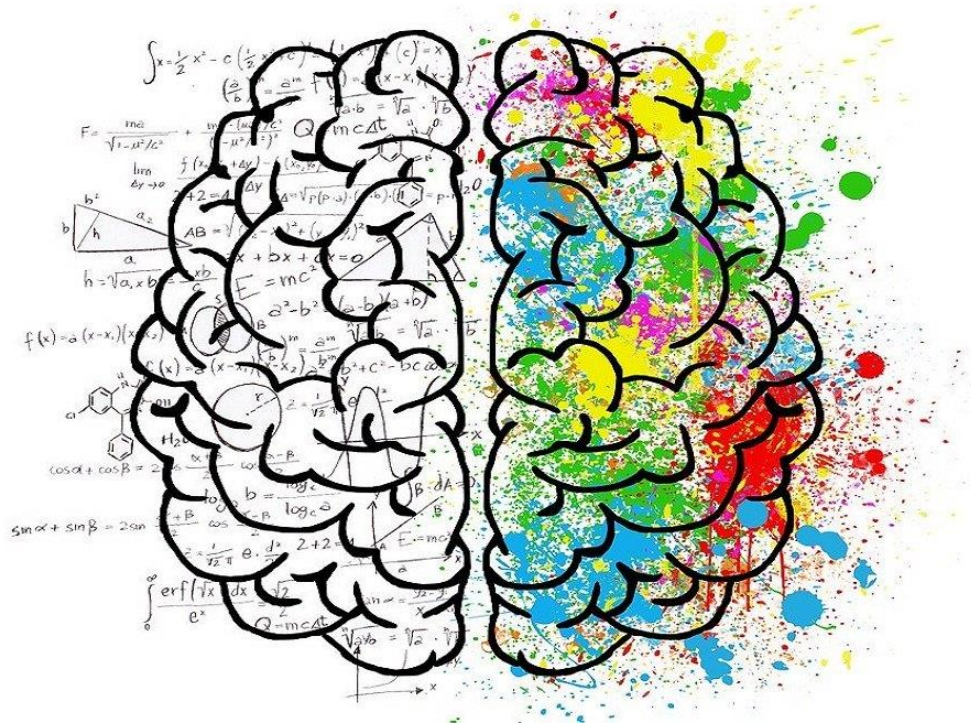
построенная по принципу организации и функционирования

биологических нейронных сетей — сетей нервных клеток живого организма.

Это понятие возникло при изучении процессов, протекающих в мозге.



Мозг

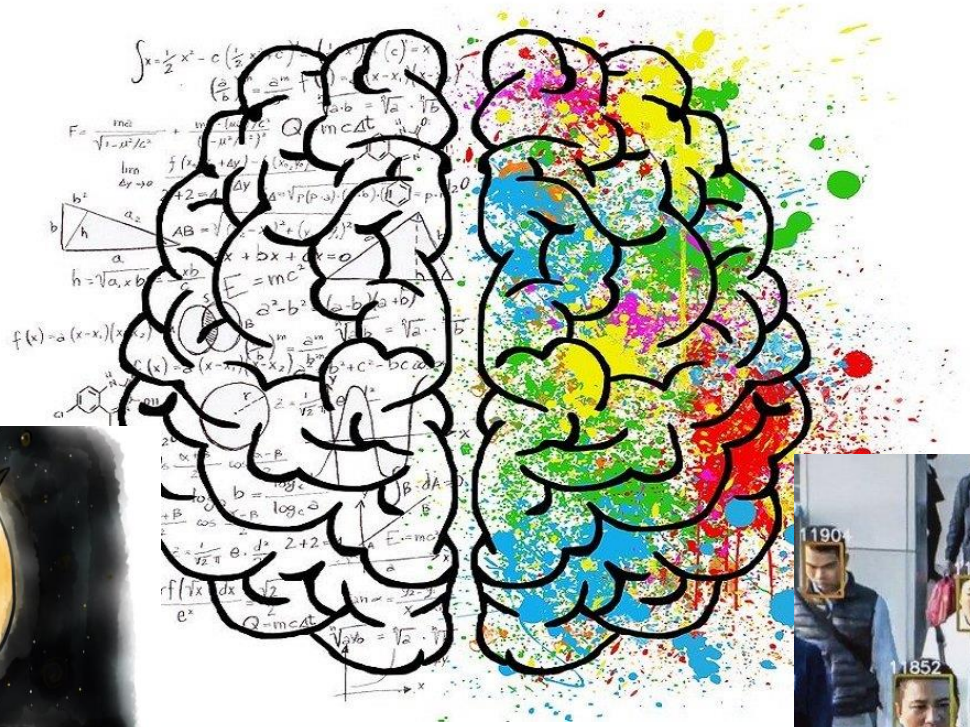


Мозг – это

- Система обработки информации
- Сложный, нелинейный, параллельный компьютер

Решает множество сложных задач
из области распознавания образов, обработки сигналов и прочее

Почему мы справляемся со столь сложными задачами?



Почему мы справляемся со столь сложными задачами?

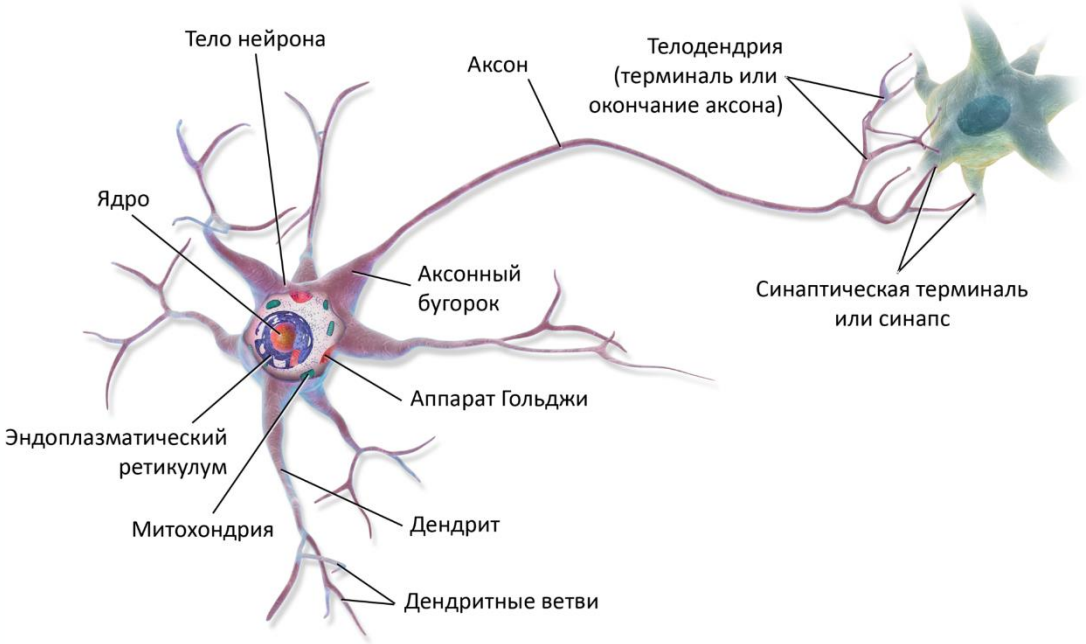
Опыт → Нейронная сеть → Способность решать сложные задачи

Обучение / training

Применение / inference

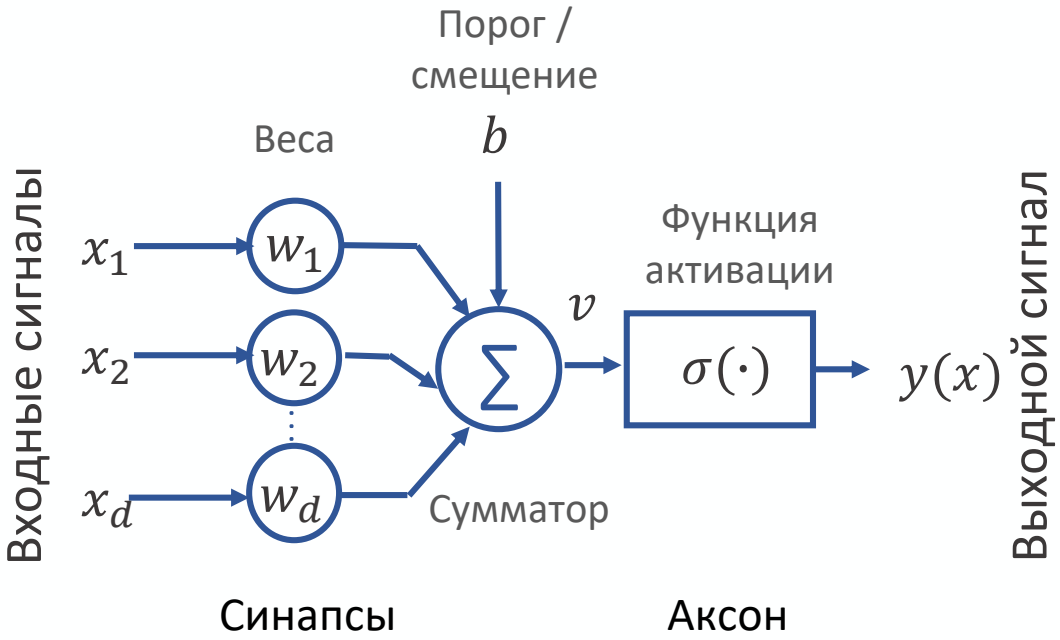
Бионика: нейронные сети

Биологический нейрон



Нейрон накапливает заряд,
и если заряд больше порога,
то нейрон подает сигнал дальше.

Искусственный нейрон

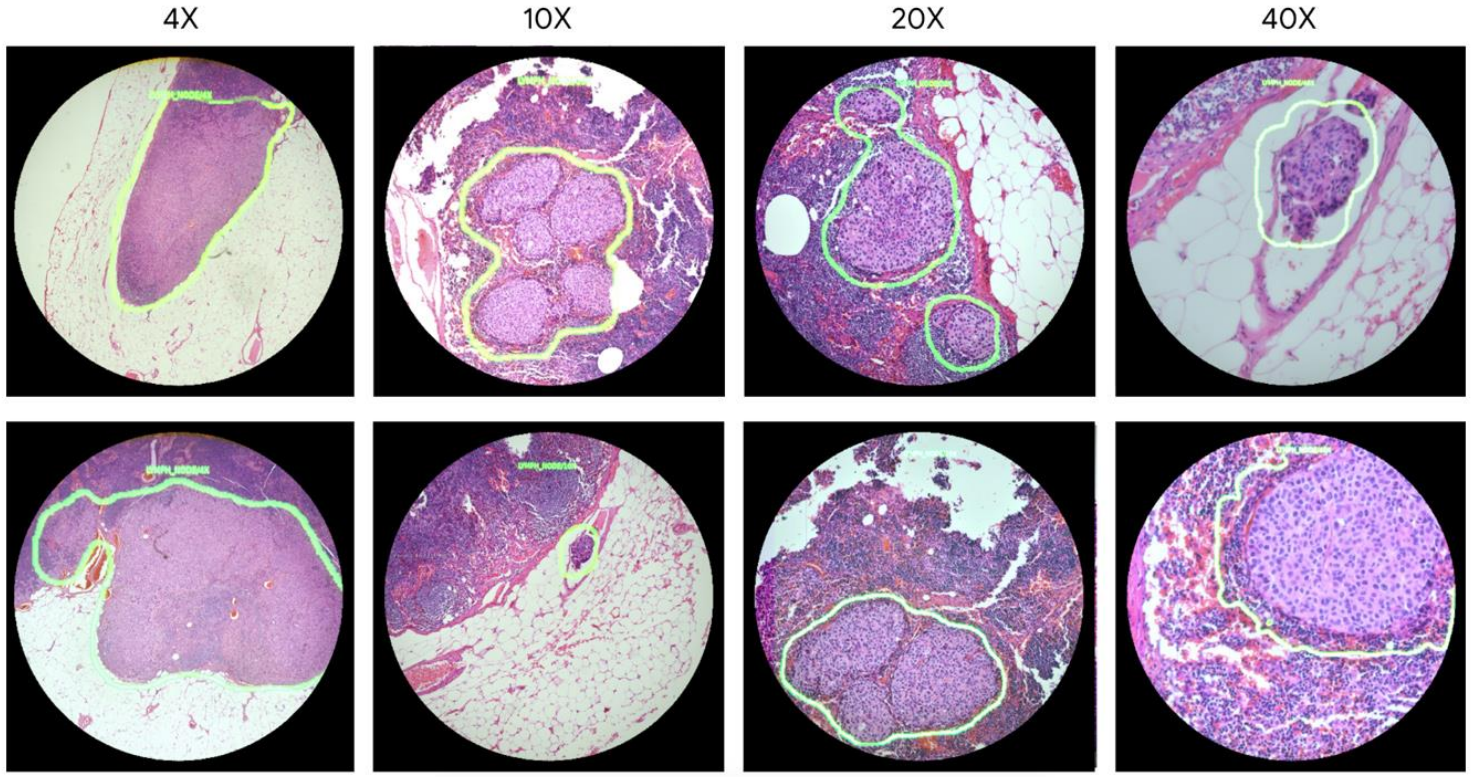


Модель МакКаллока-Питца,
где σ — пороговая функция активации

Области применения

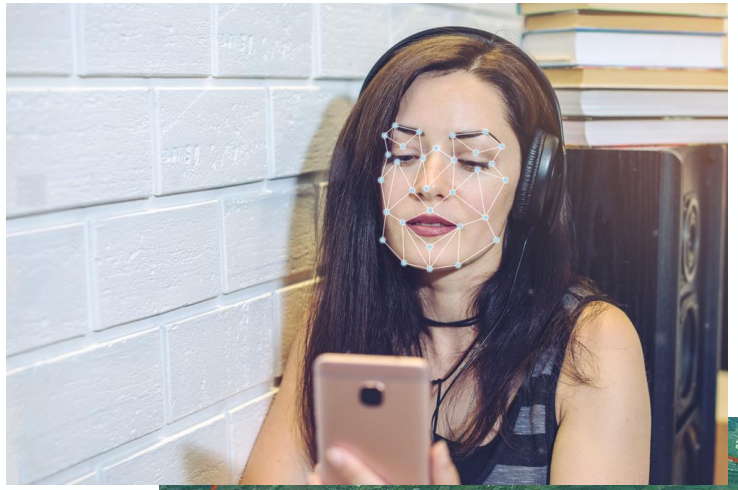
Сегментация опухоли / Tumor segmentation

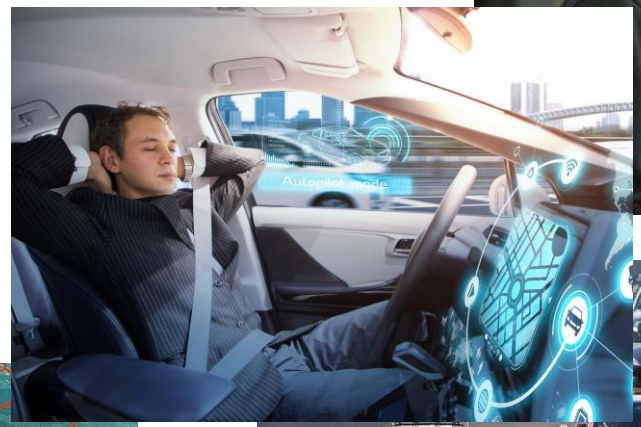
Цель — для каждого пикселя определить принадлежит ли он опухоли или нет.

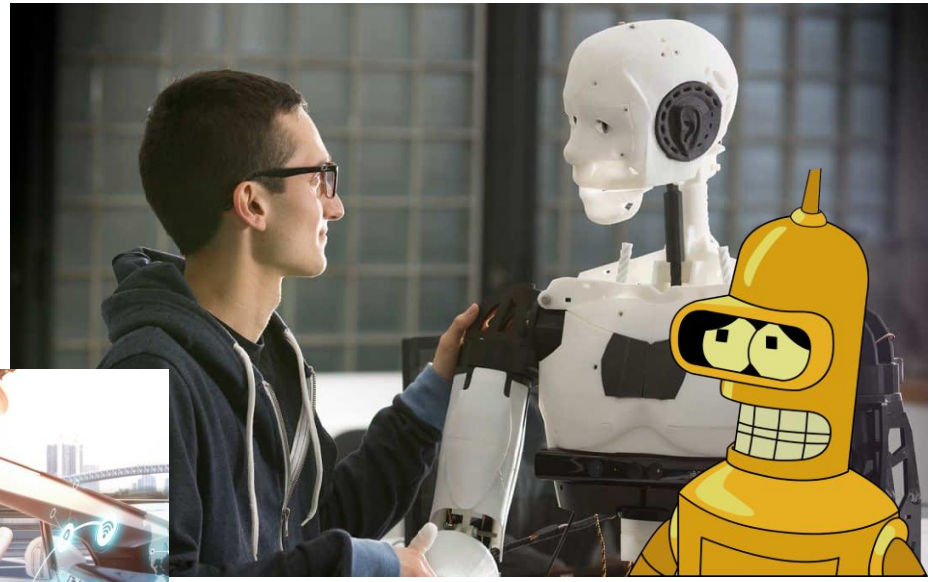


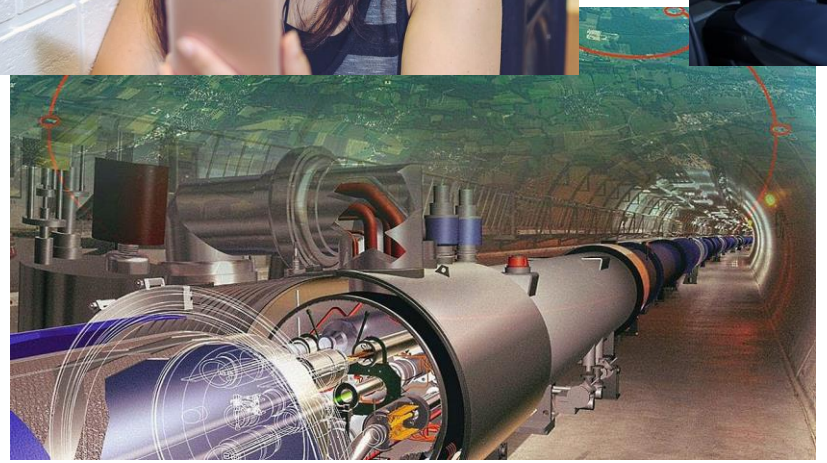
Области применения

И многое другое...









Модель нейрона

Обозначим

$x = (x_1, \dots, x_d)^T \in \mathbb{R}^d$ — один объект, где $x_1, \dots, x_d$ — признаки;

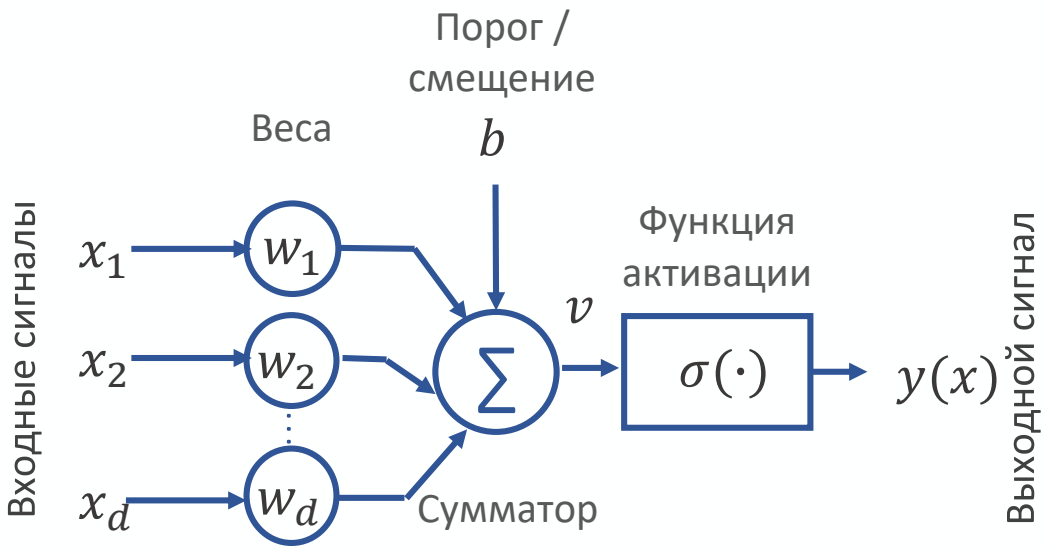
$w = (w_1, \dots, w_d)^T \in \mathbb{R}^d$ — вектор весов;

$b \in \mathbb{R}$ — смещение.

Выход нейрона —

$$y(x) = \sigma(\langle x, w \rangle + b) = \sigma\left(\sum_{j=1}^{d} w_j x_j + b\right),$$

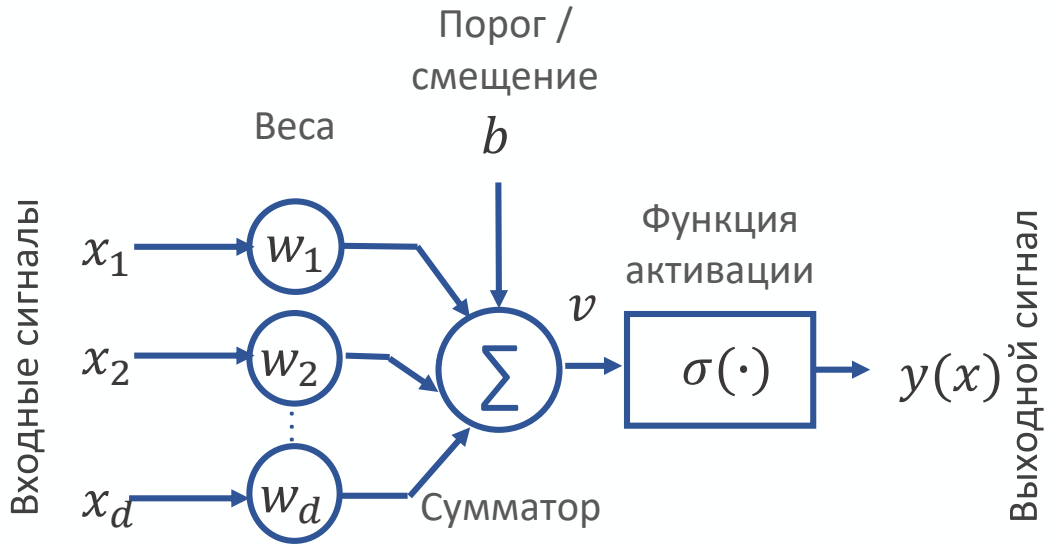
где σ — некоторая кус.-дифф. функция, назовем ее **функцией активации**.



Что-то знакомое...

На что похожа эта формула

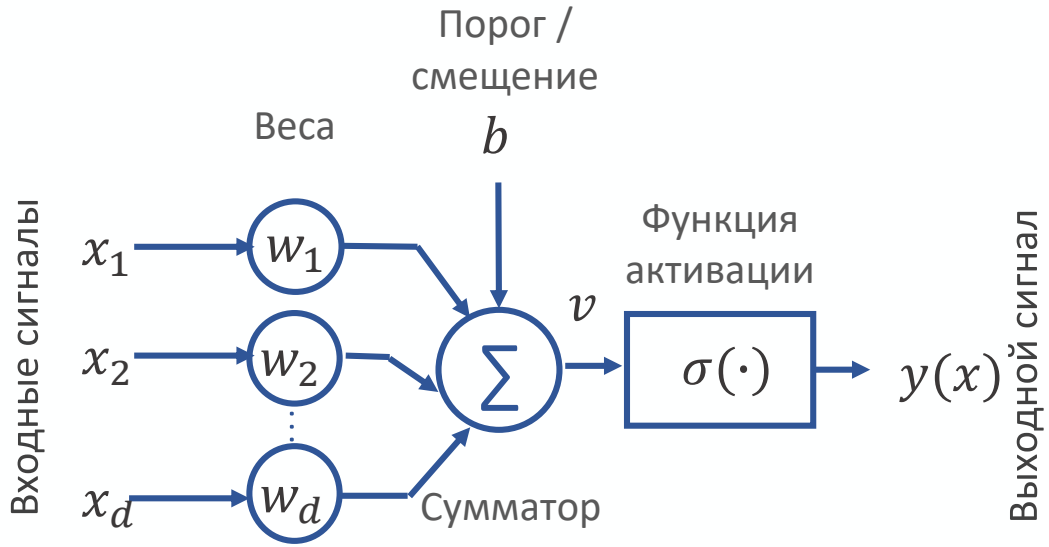
$$y(x) = \sigma(\langle x, w \rangle + b) = \sigma\left(\sum_{j=1}^{d} w_j x_j + b\right)$$



Что-то знакомое...

На что похожа эта формула?

$$y(x) = \sigma(\langle x, w\rangle + b) = \sigma\left(\sum_{j=1}^{d} w_j x_j + b\right)$$



Линейная регрессия для 1 эл.

$$y(x) = \langle x, w\rangle + b = \sigma(\langle x, w\rangle + b) = \sigma\left(\sum_{j=1}^{d} x_j w_j + b\right), \qquad \text{где } \sigma(z) = z -$$

линейная ф-я.

Логистическая регрессия для 1 эл.

$y(x) = \sigma(\langle x, w\rangle + b) = \sigma\left(\sum_{j=1}^{d} x_j w_j + b\right)$, где $(z) = \frac{1}{1+e^{-z}}$ — логистическая сигмоида.

Пуассоновская регрессия для 1 эл.

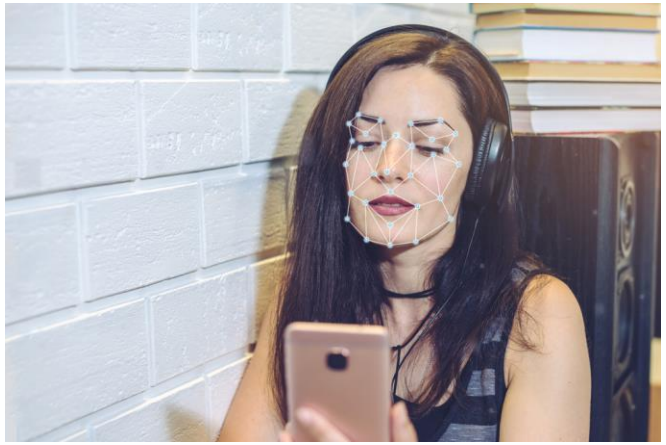
$y(x) = \sigma(\langle x, w\rangle + b) = \sigma\left(\sum_{j=1}^{d} x_j w_j + b\right)$, где $(z) = e^z$.

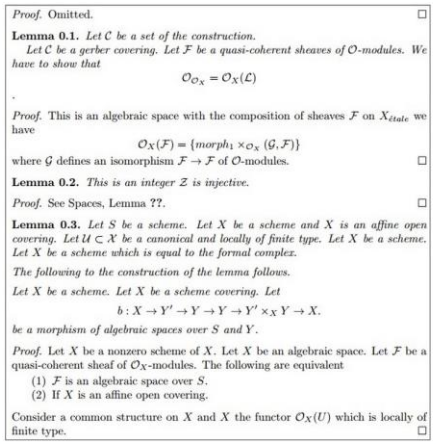
Более сложные задачи

- У линейной и логистической регрессий ограниченная область применения.
- Для того, чтобы решить нелинейную задачу,
  нужно делать сложные преобразования с признаками.
- Один нейрон не справится со сложными задачами... ☹

Вспомним, что в нервной системе очень много нейронов.

Значит, будем использовать **больше нейронов**!





Proof. Omitted. □

Lemma 0.1. *Let $\mathcal{C}$ be a set of the construction.*
Let $\mathcal{C}$ be a gerber covering. Let $\mathcal{F}$ be a quasi-coherent sheaves of $\mathcal{O}$-modules. We have to show that

$$\mathcal{O}_{\mathcal{O}_X} = \mathcal{O}_X(\mathcal{L})$$

.

Proof. This is an algebraic space with the composition of sheaves $\mathcal{F}$ on $X_{\acute{e}tale}$ we have

$$\mathcal{O}_X(\mathcal{F}) = \{morph_1 \times_{\mathcal{O}_X} (\mathcal{G}, \mathcal{F})\}$$

where $\mathcal{G}$ defines an isomorphism $\mathcal{F} \to \mathcal{F}$ of $\mathcal{O}$-modules. □

Lemma 0.2. *This is an integer $\mathcal{Z}$ is injective.*

Proof. See Spaces, Lemma ??. □

Lemma 0.3. *Let S be a scheme. Let X be a scheme and X is an affine open covering. Let $\mathcal{U} \subset \mathcal{X}$ be a canonical and locally of finite type. Let X be a scheme. Let X be a scheme which is equal to the formal complex.*

The following to the construction of the lemma follows.

Let X be a scheme. Let X be a scheme covering. Let

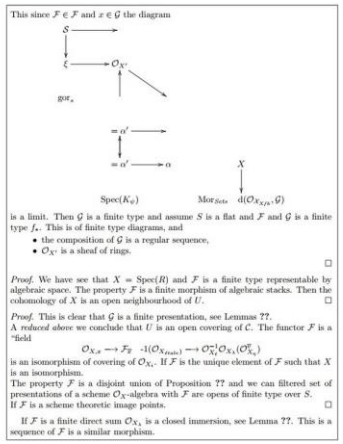
$$b : X \to Y' \to Y \to Y \to Y' \times_X Y \to X.$$

be a morphism of algebraic spaces over S and Y.

Proof. Let X be a nonzero scheme of X. Let X be an algebraic space. Let $\mathcal{F}$ be a quasi-coherent sheaf of $\mathcal{O}_X$-modules. The following are equivalent

(1) $\mathcal{F}$ is an algebraic space over S.
(2) If X is an affine open covering.

Consider a common structure on X and X the functor $\mathcal{O}_X(U)$ which is locally of finite type. □



This since $\mathcal{F} \in \mathcal{F}$ and $x \in \mathcal{G}$ the diagram

$$\begin{array}{ccc} S & & \\ \downarrow \xi & \to & \mathcal{O}_{X'} \\ & \searrow^{\text{gor}_s} & \\ & & \end{array} \qquad \begin{array}{ccc} = \alpha' & & \\ \downarrow & & \\ = \alpha' & \to & \alpha \end{array} \qquad \begin{array}{c} X \\ \downarrow \\ \end{array}$$

$$\text{Spec}(K_\psi) \qquad \text{Mor}_{\text{Sets}} \quad \mathrm{d}(\mathcal{O}_{X_{\acute{e}t}}, \mathcal{G})$$

is a limit. Then $\mathcal{G}$ is a finite type and assume S is a flat and $\mathcal{F}$ and $\mathcal{G}$ is a finite type f_*. This is of finite type diagrams, and

- the composition of $\mathcal{G}$ is a regular sequence,
- $\mathcal{O}_{X'}$ is a sheaf of rings.

□

Proof. We have see that $X = \text{Spec}(R)$ and $\mathcal{F}$ is a finite type representable by algebraic space. The property $\mathcal{F}$ is a finite morphism of algebraic stacks. Then the cohomology of X is an open neighbourhood of U. □

Proof. This is clear that $\mathcal{G}$ is a finite presentation, see Lemmas ??.
A *reduced* above we conclude that U is an open covering of $\mathcal{C}$. The functor $\mathcal{F}$ is a "field

$$\mathcal{O}_{X,\bar{s}} \longrightarrow \mathcal{F}_{\overline{y}} \quad \text{-1}(\mathcal{O}_{X_{\acute{e}tale}}) \longrightarrow \mathcal{O}_{X_i}^{-1}\mathcal{O}_{X_i}(\mathcal{O}_{X_0}^*)$$

is an isomorphism of covering of $\mathcal{O}_{X_i}$. If $\mathcal{F}$ is the unique element of $\mathcal{F}$ such that X is an isomorphism.
The property $\mathcal{F}$ is a disjoint union of Proposition ?? and we can filtered set of presentations of a scheme $\mathcal{O}_X$-algebra with $\mathcal{F}$ are opens of finite type over S.
If $\mathcal{F}$ is a scheme theoretic image points. □

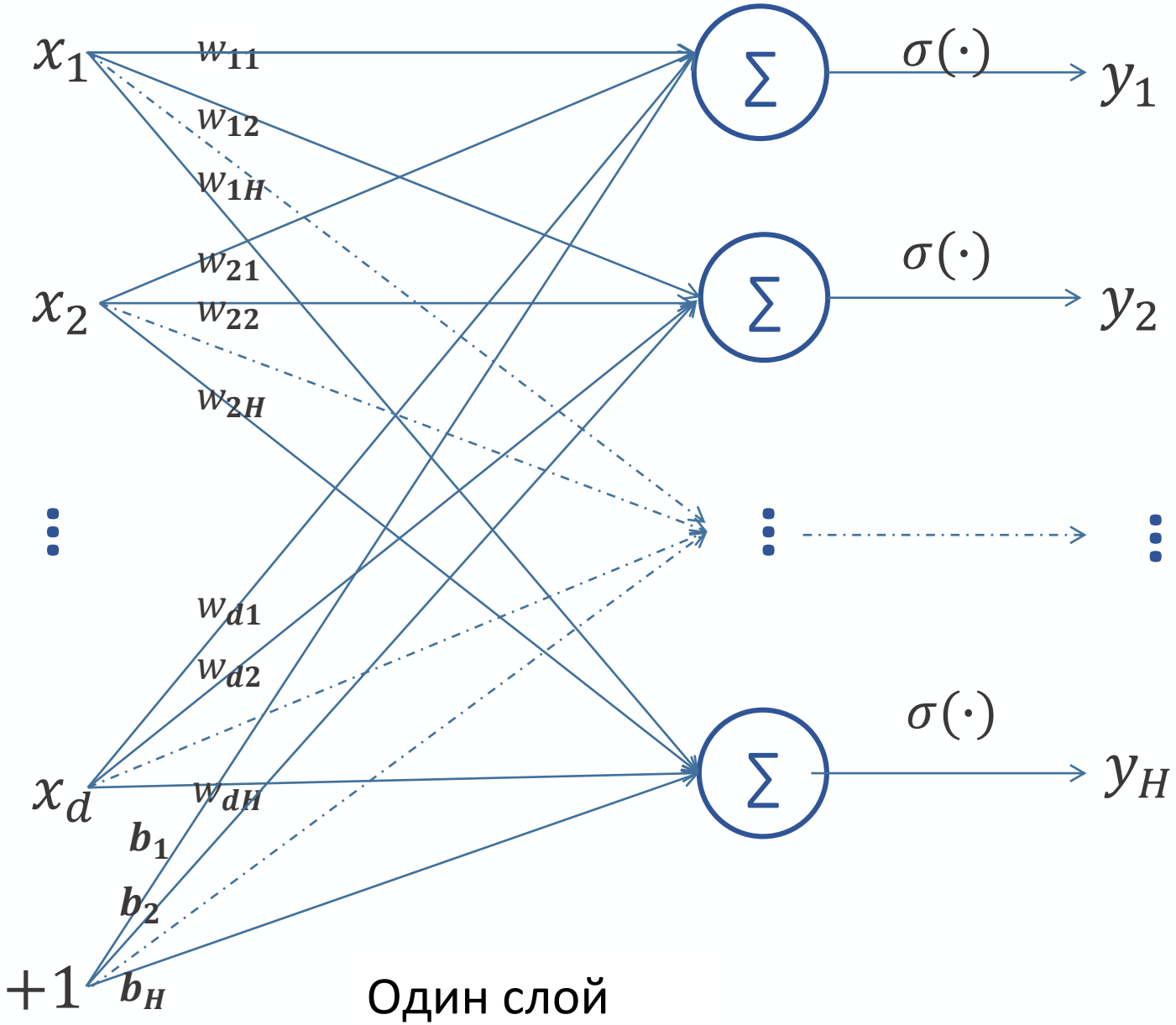
If $\mathcal{F}$ is a finite direct sum $\mathcal{O}_{X_i}$ is a closed immersion, see Lemma ??. This is a sequence of $\mathcal{F}$ is a similar morphism.

Однослойная нейронная сеть

Слой размера H
— набор из H нейронов.

$(w_{jh})_{jh} \in \mathbb{R}^{?}$ и $(b_h)_h$
— параметры модели

Такую нейронную сеть (слой нейронной сети) называют **полносвязной**(ым)



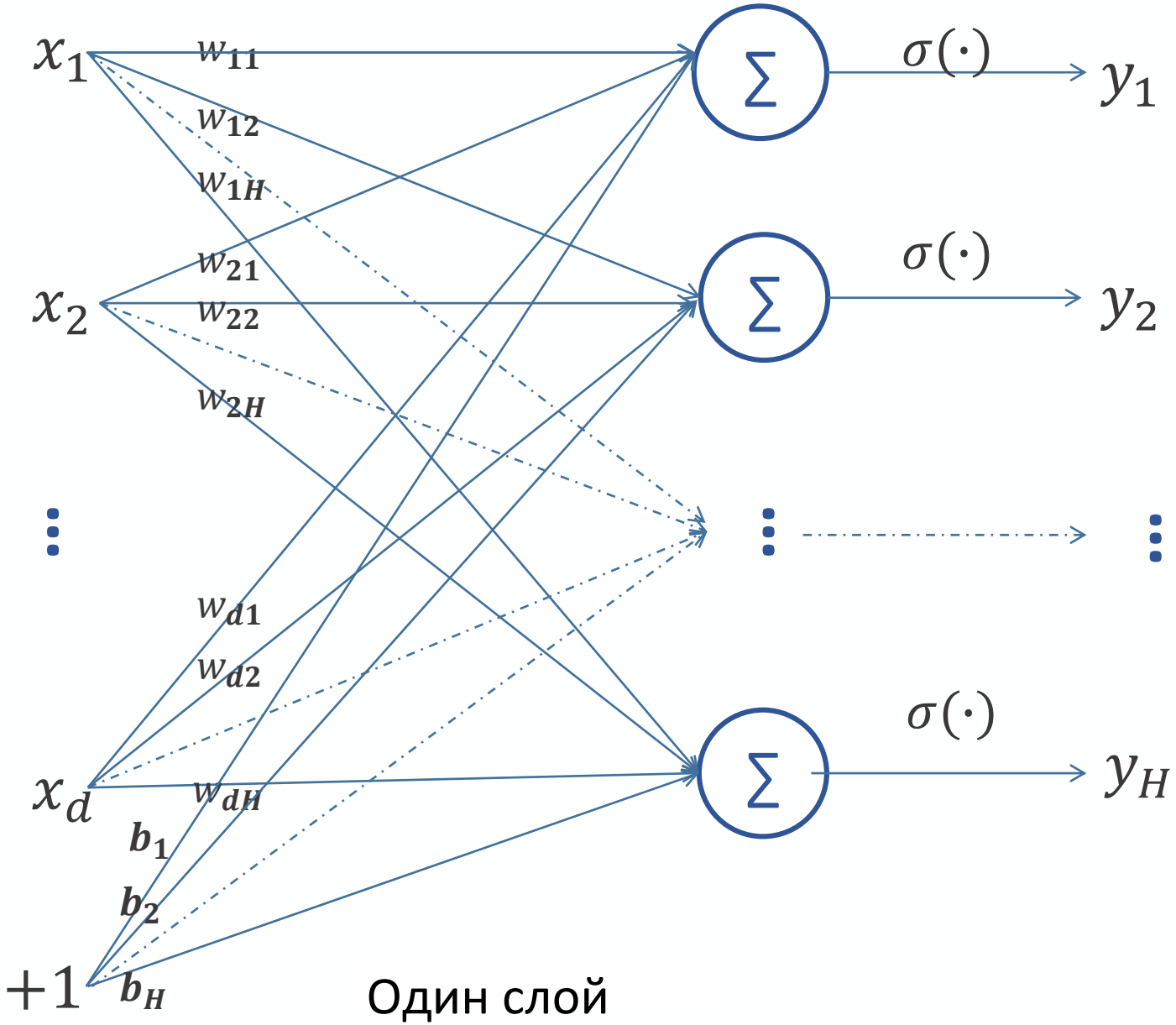
Один слой

Однослойная нейронная сеть

Слой размера H
— набор из H нейронов.

$(w_{jh})_{jh} \in \mathbb{R}^{d \times H}$ и $(b_h)_h$
— параметры модели

Такую нейронную сеть (слой нейронной сети) называют **полносвязной**(ым)



Один слой

Однослойная нейронная сеть

Матричное представление

Пусть $x = (x_1, x_2, \ldots, x_d)$ — элемент выборки.

$s = (s_1, s_2, \ldots, s_H)$ — выходы нейронов до применения функции активации.

$y = (y_1, y_2, \ldots, y_H)$ — выход слоя.

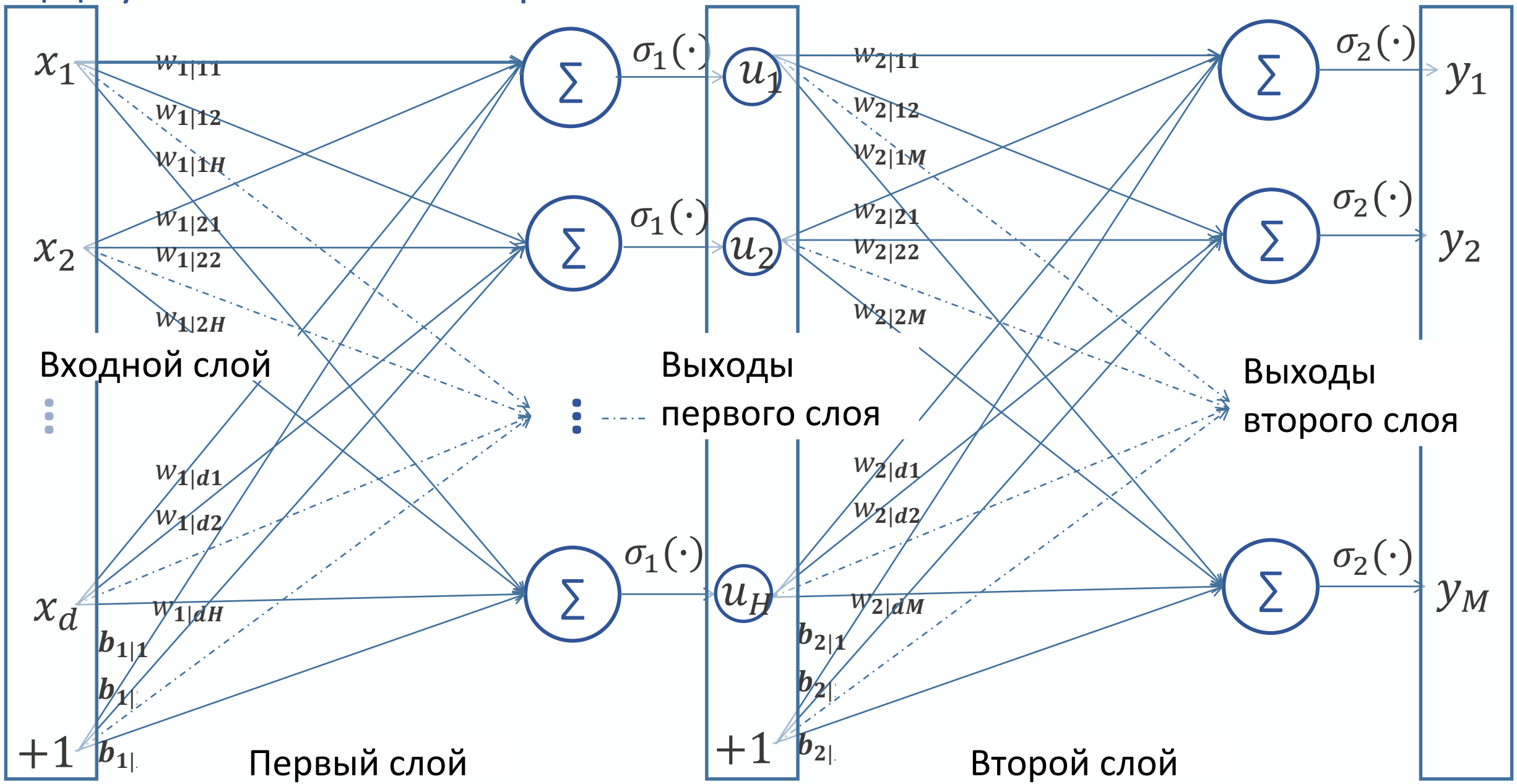
Тогда работу слоя можно описать операциями:

$$1)\ s = (x_1, x_2, \cdots, x_d) \times \begin{pmatrix} w_{11} & \cdots & w_{1h} & \cdots & w_{1H} \\ w_{21} & \cdots & w_{2h} & \cdots & w_{2H} \\ \cdots & \cdots & \cdots & \cdots & \cdots \\ w_{d1} & \cdots & w_{dh} & \cdots & w_{dH} \end{pmatrix} + \begin{pmatrix} b_1 \\ b_2 \\ \cdots \\ b_H \end{pmatrix}^T$$

$$= x^T \cdot W + b^T$$

$$2)\ y = (y_1, y_2, \ldots, y_H) = (\sigma(s_1), \sigma(s_2), \ldots, \sigma(s_H)) = \sigma(s) = \boxed{\sigma(x^T W + b^T)}$$

Двухслойная нейронная сеть



Двухслойная нейронная сеть

Матричное представление

Пусть $x = (x_1, x_2, \dots, x_d)$ — элемент выборки,

$u = (s_1, s_2, \dots, s_H)$ — выход I слоя, $y = (y_1, y_2, \dots, y_M)$ — выход II слоя,

$W_1 = (w_{1|jh})_{jh}$ — м-ца весов I слоя, $W_2 = (w_{2|hm})_{hm}$ — м-ца весов II слоя,

$b_1 = (b_{1|1}, \dots, b_{1|H})^T$ — в-р сдвигов I слоя, $b_2 = (b_{2|1}, \dots, b_{2|H})^T$ — в-р сдвигов II слоя.

Тогда работу двухслойной нейронной сети можно представить как:

1) $u = \sigma_1(x^T W_1 + {b_1}^T)$

2) $y = \sigma_2\left(u^T W_2 + {b_2}^T\right) = \boxed{\sigma_2\left(\sigma_1\left(x^T W_1 + {b_1}^T\right)^T W_2 + {b_2}^T\right)}$

Двухслойная нейронная сеть

Назовем функцию $\sigma(z)$ сигмоидой, если $\lim\limits_{z\to-\infty}(z) = 0$ и $\lim\limits_{z\to+\infty}\sigma(z) = 1$.

$\sigma(z) = \frac{e^z}{1+e^z}$ — логистическая сигмоида, один из примеров такой функции.

Теорема (Цыбенко, 1989)

Если $\sigma(z)$ - непрерывная сигмоида, то для любой непрерывной на $[0,1]^d$ функции $f(x)$

существуют такое H и значения параметров $w_{1,h} \in \mathbb{R}^d, w_{2,h} \in \mathbb{R}^d, b \in \mathbb{R}$,

что двухслойная нейросеть $g(x) = \sum\limits_{h=1}^{H} w_{2,h} \cdot \sigma(x^T w_{1,h} + b)$

равномерно приближает $f(x)$ с любой точностью ε:

$|g(x) - f(x)| < \varepsilon$, для всех $x \in [0,1]^d$

George Cybenko. Approximation by Superpositions of a Sigmoidal functions. Mathematics of Control, Signals, and Systems. 1989.

Двухслойная нейронная сеть

Выводы

- С помощью линейных операций и функций активаций σ от одного аргумента можно вычислять **любую непрерывную функцию** на заданном интервале с любой желаемой точностью.
- **Двух слоев** в нейронной сети **теоретически достаточно**.

Замечания

- Теорема ничего не говорит о количестве нейронов в каждом слое, о значении весов и сдвигов, и виде функции активации.
- Двумя слоями такая цель теоретически достигается, но сложно.

Только в одной коре головного мозга число слоев равно 6.

- Дополнительные слои - удобный способ преобразования признаков, переход из одного признакового пространства в более удобное для решения задачи.

Нейронная сеть

Мы параметризовали модель нейронной сети.

Из теоремы Цыбенко вытекает, что существую параметры, при которых

мы сможем аппроксимировать любую непр. функцию на заданном интервале.

Хотим автоматически находить параметры сети!

А как?

Вспомним линейную регрессию...

Решение задачи регрессии

Линейная регрессия: $\hat{y} = Xw + b$,т.е. $\hat{y}_i = \sum_{j=1}^{d} x_{ij} w_j + b$

где $X = (x_{ij})_{ij}$ — матрица входных данных, $i \in \{1, \dots, n\}, j \in \{1, \dots, d\}$

$\hat{y} = (\hat{y}_1, \dots, \hat{y}_d)^T$ — вектор предсказания,

$w = (w_1, \dots, w_d)^T$ — вектор весов, b — сдвиг.

Задачу можно решить аналитически. А можно с помощью **градиентного спуска**.

Зададим функцию, которую мы хотим минимизировать

$$L = \frac{1}{n} \sum_{i=1}^{n} (\hat{y}_i - y_i)^2$$ — MSE (Mean Squared Error)

Пусть $\theta = (w, b)$, тогда оптимизация будет следующей:

$$\theta_t = \theta_{t-1} - \eta \nabla L(\theta_t),$$ где η — скорость обучения

Обучение нейронной сети

Обозначим все параметры сети как θ.

Пусть $\mathcal{L}(\hat{y}_\theta,\ y)$ — **функция потерь** на объекте x.

Она сравнивает предсказания сети $\hat{y}_\theta$ с откликом y на объекте x.

Минимизируем **эмпирический риск** по обучающей выборке $x_1, \ldots, x_n$:

$$Q(w) = \frac{1}{n}\sum_{i=1}^{n}\mathcal{L}(\hat{y}_{\theta,\mathrm{i}}, y_i) \ \rightarrow \min_\theta$$

Решаем задачу минимизации с помощью **градиентного спуска**

$$\theta_t = \theta_{t-1} - \eta\nabla Q(\theta_t), \quad \text{где } \eta \text{ — скорость обучения}$$

Обучение нейронной сети

Задача

Посчитать градиенты функции потерь по всем параметрам.

Наивный подход

- Посчитать для каждого параметра градиент отдельно.
- Подсчет одного градиента линеен по количеству параметров.

⇒ сложность такой процедуры **квадратична** по количеству параметров.

Можно ускорить!

Что-то новенькое

Метод обратного распространения ошибки

Метод вытекает из **формулы производной сложной функции**.

Если $f(x) = g_m(g_{m-1}(\dots(g_1(x))\dots))$

то $$\frac{\partial f}{\partial x} = \frac{\partial g_m}{\partial g_{m-1}} \frac{\partial g_{m-1}}{\partial g_{m-2}} \cdots \frac{\partial g_2}{\partial g_1} \frac{\partial g_1}{\partial x}$$

Градиенты будем вычислять последовательно
от выхода нейронной сети к входу, начиная с $\frac{\partial g_m}{\partial g_{m-1}}$
и умножая каждый раз на частные производные предыдущего слоя.

Тогда сложность будет **линейной**.

Метод обратного распространения ошибки

Пусть x_i – вход i – го слоя сети, а O_i – его выход. Градиент $\mathcal{L}(\hat{y}_i, y_i)$ по весам

$$\frac{\partial \mathcal{L}(\hat{y}_i, y_i)}{\partial W_m} = \frac{\partial \mathcal{L}(\hat{y}_i, y_i)}{\partial O_m} \frac{\partial O_m}{\partial W_m}$$

Зависит только от вида слоя, можем закодить заранее

Легко считается для последнего слоя, но не для остальных

Заметим, что $x_{i+1} = O_i$

$$\frac{\partial \mathcal{L}(\hat{y}_i, y_i)}{\partial O_m} = \frac{\partial \mathcal{L}(\hat{y}_i, y_i)}{\partial x_{m+1}} = \frac{\partial \mathcal{L}(\hat{y}_i, y_i)}{\partial O_{m+1}} \frac{\partial O_{m+1}}{\partial x_{m+1}}$$

Зависит только от вида слоя, можем закодить заранее

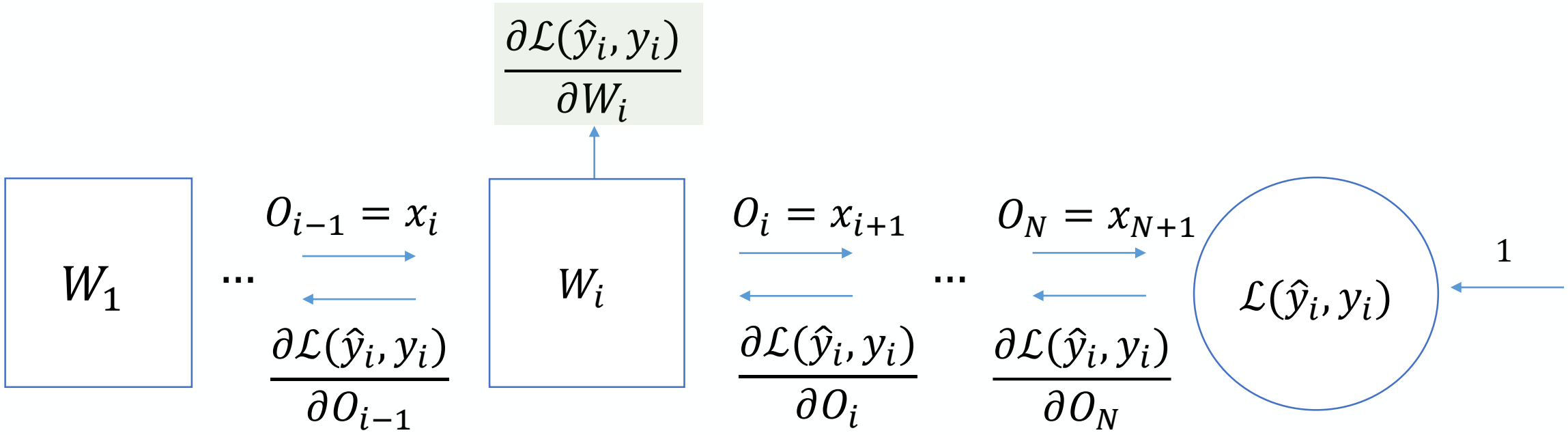
Пересчитываем предыдущий слой через следующий!
Стартуем с последнего слоя, где легко считаем

Метод обратного распространения ошибки

Пусть x_i – вход $i-$го слоя сети, а O_i – его выход. Градиент $\mathcal{L}(\hat{y}_i, y_i)$ по весам

$$\frac{\partial \mathcal{L}(\hat{y}_i, y_i)}{\partial W_m} = \frac{\partial \mathcal{L}(\hat{y}_i, y_i)}{\partial O_m} \frac{\partial O_m}{\partial W_m}$$

$$\frac{\partial \mathcal{L}(\hat{y}_i, y_i)}{\partial O_m} = \frac{\partial \mathcal{L}(\hat{y}_i, y_i)}{\partial x_{m+1}} = \frac{\partial \mathcal{L}(\hat{y}_i, y_i)}{\partial O_{m+1}} \frac{\partial O_{m+1}}{\partial x_{m+1}}$$



Обучение нейронной сети

Итоговый алгоритм

1. Инициализировать все веса.
2. Повторить NUMBER_OF_STEPS раз:
    2.1 Прямое распространение (Forward pass) — вычисляем выходы всех нейронов

    2.2 Вычисляем ошибку предсказания (Loss)

    2.3 Обратное распространение (Backward pass) — считаем последовательно производные с выхода сети до входа.

    2.4 Обновление весов (Update / Optimizer step) — обновляем веса, производя шаг оптимизации

Везде свои тонкости

Обучение нейронной сети

Подставим эмпирический риск в формулу градиентного спуска

$$\theta_t = \theta_{t-1} - \eta \nabla \left(\frac{1}{n} \sum_{i=1}^{n} \mathcal{L}(\hat{y}_{\theta_{t-1}, \mathrm{i}}, y_i) \right)$$

Нейронные сети способны обучаться на огромном количестве данных.

Если мы будем считать эмпирический риск **по всем данным сразу**,

то у нас все **может не поместиться все память**.

Разные виды градиентного спуска решают эту проблему по-своему.

Обучение нейронной сети

Виды градиентного спуска

- **Batch Gradient Descent**

$$\theta_t = \theta_{t-1} - \eta\nabla\left(\frac{1}{n}\sum_{i=1}^{n}\mathcal{L}(\hat{y}_{\theta_{t-1},\mathrm{i}}, y_i)\right)$$

Разбиваем данные на блоки (батчи).

Для каждого блока считаем градиенты и накапливаем их.

Производим обновление параметров после подсчета градиента по всем данным.

- **Mini-batch Gradient Descent**

$$\theta_t = \theta_{t-1} - \eta\nabla\left(\frac{1}{b}\sum_{k=1}^{b}\mathcal{L}(\hat{y}_{\theta_{t-1},\mathrm{i}_k}, y_{i_k})\right)$$

Разбиваем данные на блоки (батчи).

Для каждого блока считаем градиент и обновляем параметры.

$(x_{i_1}, \ldots, x_{i_b})$ — текущий батч. Самый популярный метод.

- **Стохастический градиентный спуск**

$$\theta_t = \theta_{t-1} - \eta\nabla\left(\mathcal{L}(\hat{y}_{\theta_{t-1},\mathrm{i}}, y_i)\right)$$

Выбираем один элемент x_i случайно.

Считаем градиент и производим обновление.

Эквивалентно Mini-Batch GD для размера батча, равного 1.

Функции активации

Функции активации

Зачем нужны функции активации?

- Для того, чтобы делать нелинейные преобразования.
- Кроме того, по теореме Цыбенко, используя функции активации типа сигмоиды мы можем аппроксимировать любую функцию используя двухслойную нейронную сеть.

Функции активации

Сигмоида

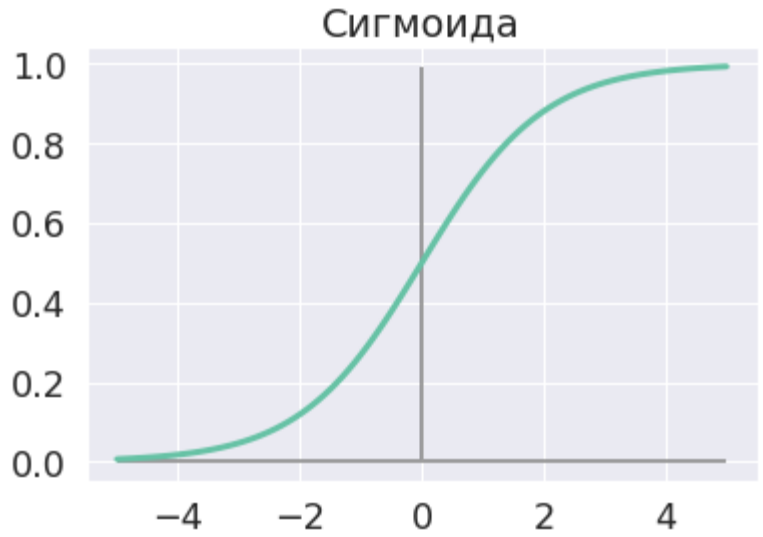
$$\sigma(z) = \frac{1}{1 + e^{-z}}$$

«+»:

- гладкий градиент, уменьшает скачки в значениях
- ограниченный диапазон значений — [0, 1]
- позволяет получать ~ вероятности

«-»:

- затухающий градиент!
- вычислительно сложно



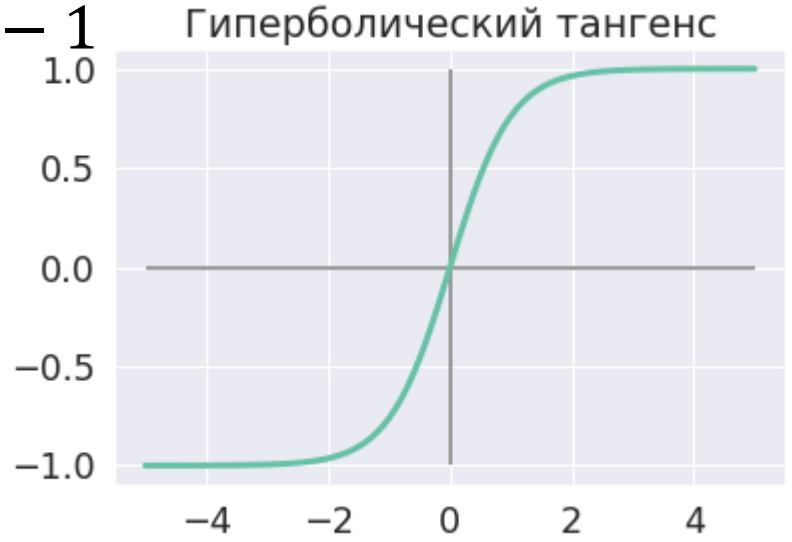
Гиперболический тангенс

$$tanh(z) = \frac{e^{z} - e^{-z}}{e^{z} + e^{-z}} = 2\sigma(2z) - 1$$

Те же плюсы и недостатки, но еще

«+»:

- ограниченный диапазон значений — [-1, 1]
- большая амплитуда градиента вблизи нуля
- лучше сигмоиды, когда не нужна нормализация [0, 1]



Функции активации

ReLU $relu(z) = max(0, z)$

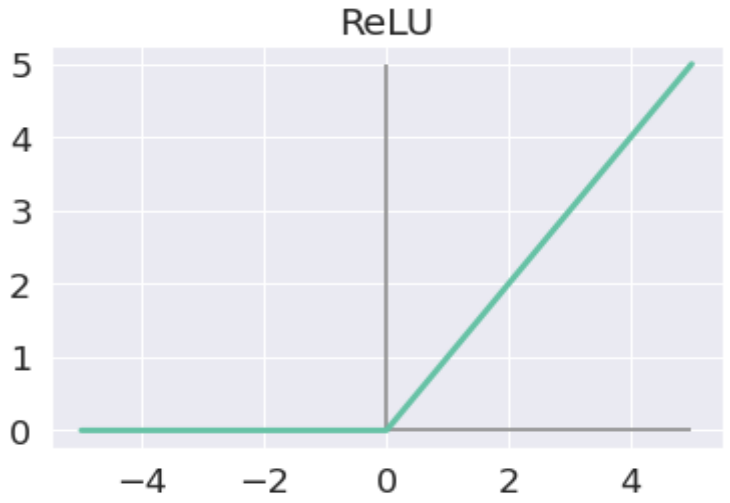
«+»:

- вычислительно просто
- ускоряет сходимость за счет выключения нейронов

«-»:

- The Dying ReLU problem

нулевые градиенты ведут к тому, что часть сети не обучается, может перестать обучаться вся сеть



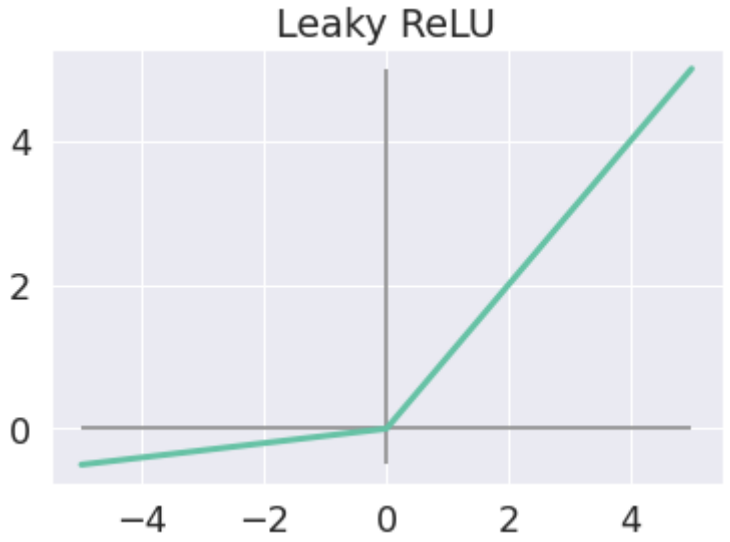
Leaky ReLU $lrelu(z) = z \cdot I\{z > 0\} + \alpha z \cdot I\{z \leq 0\}$

«+»:

- Устраняет The Dying ReLU problem.
- Сходимость все равно быстрая.

«-»:

- Иногда хуже ReLU



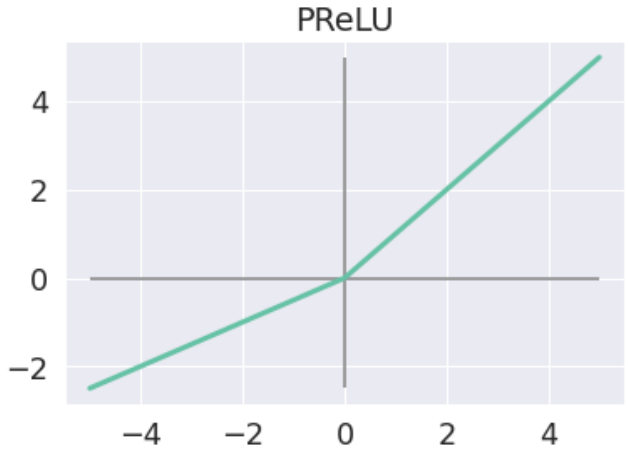
Обычно $\alpha = 0.01$

Функции активации

PReLU

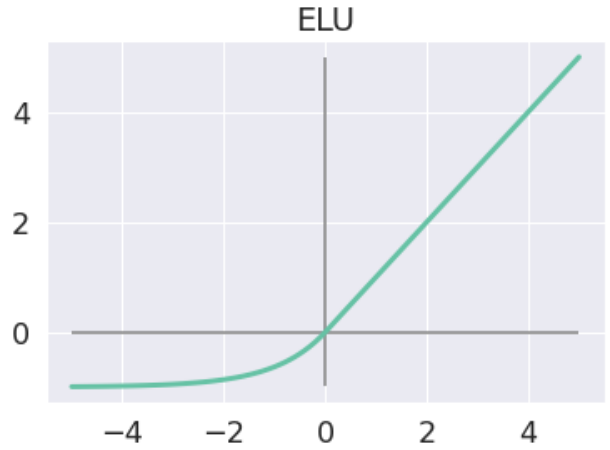
$$prelu(z) = z \cdot I\{z > 0\} + \alpha z \cdot I\{z \leq 0\}$$

Здесь α - обучаемый параметр сети.



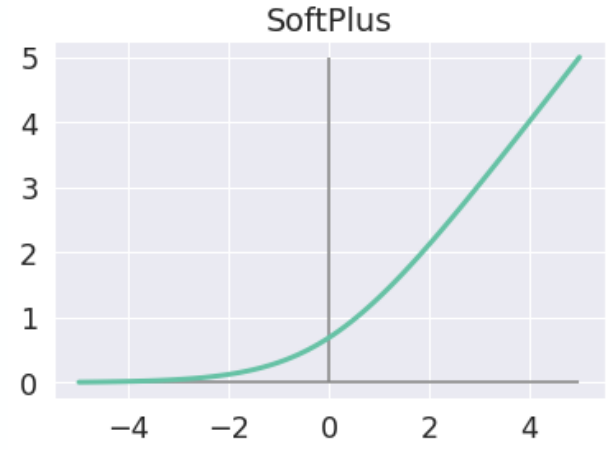
ELU

$$elu(z) = z \cdot I\{z > 0\} + \alpha(e^z - 1) \cdot I\{z \leq 0\}$$



SoftPlus

$$softplus(z) = log_e(1 + e^z)$$



Голова сети

Что предсказываем сетью?

- Обучение с учителем

  Пусть X — множество признаков, а Y — множество откликов. Нужно построить отображение $a: X \to Y$, приближающее истинную зависимость.

- Регрессия

  Пусть $Y \subset \mathbb{R}^d$ $\Rightarrow$ Последним слоем должна стоять функция, возвращающая действительные числа в нужном диапазоне.

  - $Y = \mathbb{R}^d$ $\Longrightarrow$ Можно взять линейный слой
  - $Y = [-1, 1]$ $\Longrightarrow$ Можно взять функцию активации $y(x) = \tanh(x)$

- Бинарная классификация

  Пусть $Y = \{0, 1\}$. Тогда задача сводится к задаче регрессии — предсказанию вероятности класса 1.

  - В качестве финального слоя можно брать сигмоиду (логистическая регрессия)
  - Можно брать и другие функции. Например, функцию $y(x) = 2\tanh(x) - 1$

Предсказание вероятности

Для задачи классификации часто требуется предсказать распределение вероятностей по классам.

⇒ Нужен многомерный выход.

На выходе последнего слоя без функции активации получим значения, не суммирующиеся в 1.

Чтобы отнормировать значения применяется функция активации softmax.

$softmax(z)_k = \frac{e^{z_k}}{\sum_{i=1}^{K} e^{z_i}} k \in [1, K]$, K - кол-во классов.

Отнормировали K значений на отрезок $[0,1]$ так, что они суммируются в 1.

Предсказание вероятности

Для задачи классификации часто требуется предсказать распределение вероятностей по классам.

⇒ Нужен многомерный выход.

На выходе последнего слоя без функции активации получим значения, не суммирующиеся в 1.

Чтобы отнормировать значения применяется функция активации softmax.

$softmax(z)_k = \frac{e^{z_k}}{\sum_{i=1}^{K} e^{z_i}} k \in [1, K]$, K - кол-во классов.

Отнормировали K значений на отрезок $[0,1]$ так, что они суммируются в 1.

Но почему бы просто не взять argmax из значений как метку класса?

Предсказание вероятности

Для задачи классификации часто требуется предсказать распределение вероятностей по классам.

⇒ Нужен многомерный выход.

На выходе последнего слоя без функции активации получим значения, не суммирующиеся в 1.

Чтобы отнормировать значения применяется функция активации softmax.

$softmax(z)_k = \frac{e^{z_k}}{\sum_{i=1}^{K} e^{z_i}} k \in [1, K]$, K - кол-во классов.

Отнормировали K значений на отрезок $[0,1]$ так, что они суммируются в 1.

Но почему бы просто не взять argmax из значений как метку класса?

Градиент будет нулевым практически везде, что не позволит сети нормально обучаться.

Предсказание вероятности

Для задачи классификации часто требуется предсказать распределение вероятностей по классам.

⇒ Нужен многомерный выход.

На выходе последнего слоя без функции активации получим значения, не суммирующиеся в 1.

Чтобы отнормировать значения применяется функция активации softmax.

$softmax(z)_k = \frac{e^{z_k}}{\sum_{i=1}^{K} e^{z_i}} k \in [1, K]$, K - кол-во классов.

Отнормировали K значений на отрезок $[0,1]$ так, что они суммируются в 1.

Но почему бы просто не взять argmax из значений как метку класса?

Градиент будет нулевым практически везде, что не позволит сети нормально обучаться.

Замечания

- Softmax на самом деле не является ф-й активации, т. к. принимает на вход вектор, а не скаляр.
- Простое деление на сумму не отмасштабирует значения на $[0,1]$, если какие-то из значений отриц.

Функции потерь

Функции потерь

- Регрессия

  Пусть $Y \subset \mathbb{R}^d$ $\Rightarrow$ Можно минимизировать любую норму разности. Например, MSE или MAE.

- Бинарная классификация

  Пусть $Y = \{0, 1\}$. Тогда запишем логарифм правдоподобия выборки

$$l = \sum_{i=1}^{N} (y_i \log p_i + (1 - y_i) \log(1 - p_i)) = \sum_{i=1}^{N} -\mathcal{L}(y_i, p_i)$$

  Получили функцию потерь для бинарной классификации – бинарную кросс-энтропию

$$\mathcal{L}(y_i, p_i) = -(y_i \log p_i + (1 - y_i) \log(1 - p_i))$$

Функции потерь

- Многоклассовая классификация

Пусть $Y = \{0, \dots, K\}$. Тогда запишем логарифм правдоподобия выборки

$$l = \sum_{i=1}^{N} \sum_{k=1}^{K} I\{y_i = k\} \log p_{ik} = \sum_{i=1}^{N} -\mathcal{L}(y_i, p_i)$$

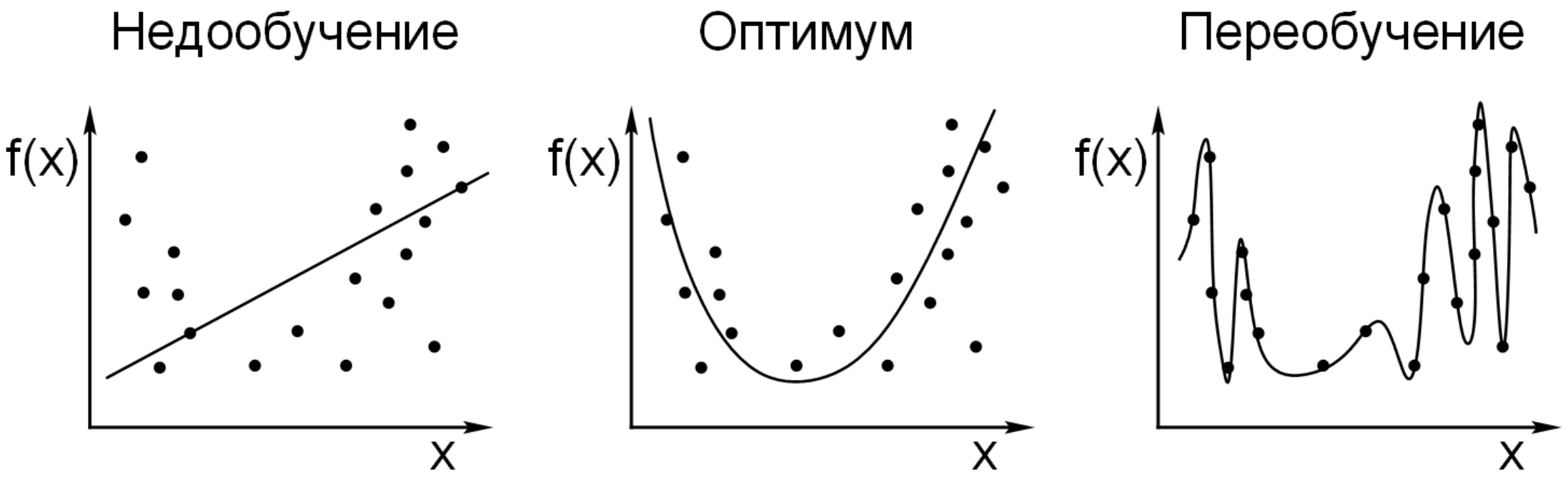
Получили общую функцию потерь для классификации – кросс-энтропию

$$\mathcal{L}(y_i, p_i) = -\sum_{k=1}^{K} I\{y_i = k\} \log p_{ik}$$

Dropout
Метод случайных отключений нейронов

Переобучение моделей

- **Переобучение** — эффект, при котором модель хорошо аппроксимирует данные на обучающей выборке, но очень плохо в общем.

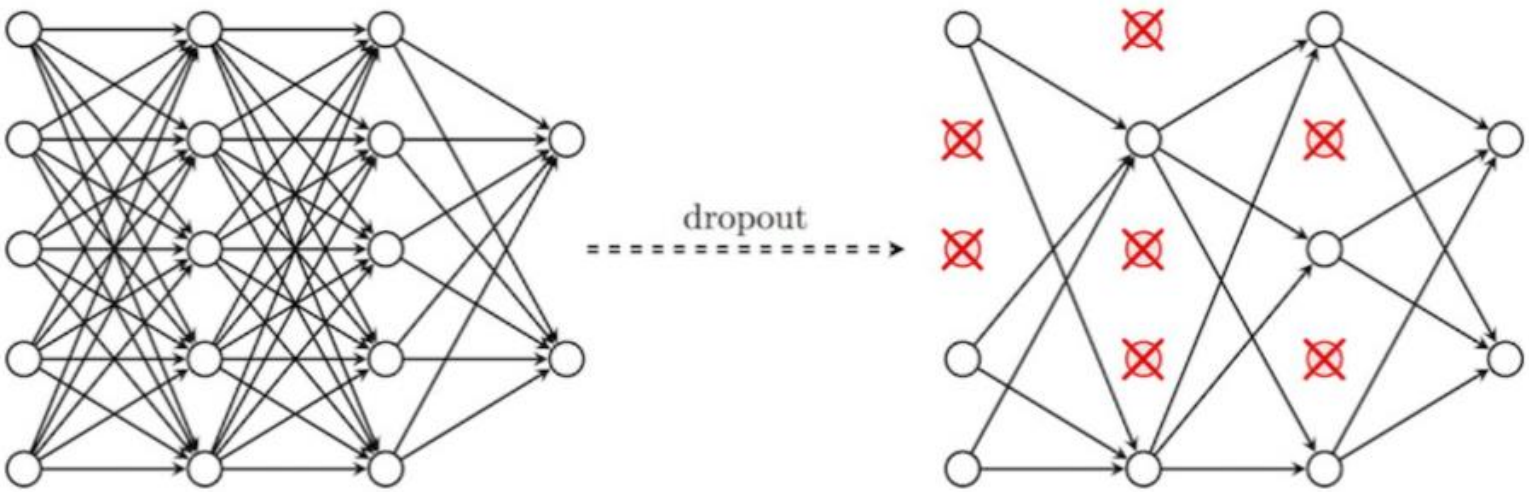


Dropout

На этапе обучения на каждой итерации выключаем каждый нейрон случайно с вероятностью p.

Исключенные нейроны не вносят вклад ни на одном из этапов обучения, в том числе при backpropagation-е (производные по ним зануляются).

На этапе тестирования включаем все нейроны.



Dropout

Является одним из способов **борьбы с переобучением**.

1. Сеть работает с частично доступными данными
и поэтому становится более устойчивой к шуму.
2. В сети с большим кол-вом нейронов нейроны начинают адаптироваться друг под друга,
коррелировать, что приводит к переобучению.
3. Dropout уменьшает совместную адаптацию и заставляет разные части сети
решать одну и ту же исходную задачу, а не подстраиваться под ошибки друг друга.

Dropout

Рассмотрим применение dropout к слою из H нейронов.
Обозначим выходы данного слоя (до отключения нейронов) как

$$\begin{pmatrix} u^1 \\ u^2 \\ \dots \\ u^h \\ \dots \\ u^H \end{pmatrix}$$

Пусть X_h — индикатор того, что h-ый нейрон включен.
Нейроны отключаются с вероятностью p : $P(X_h = 0) = p$

Тогда dropout можно представить как новый слой,
выходы которого представляются в виде
обучение: $\hat{u}_h = X_h u_h$
тестирование: $\hat{u}_h = (1-p)u_h$.

При тестировании включаются все H нейронов, а при обучении было в среднем $(1-p)H$.
Все H нейронов пойдут на вход нейронам следующего слоя,
но во время обучения следующий слой видел значения в $(1-p)$ раз меньшие,
что приведет к некорректной работе. Поэтому нужно делать масштабирование.

Dropout

Dropout как обучение ансамбля сетей

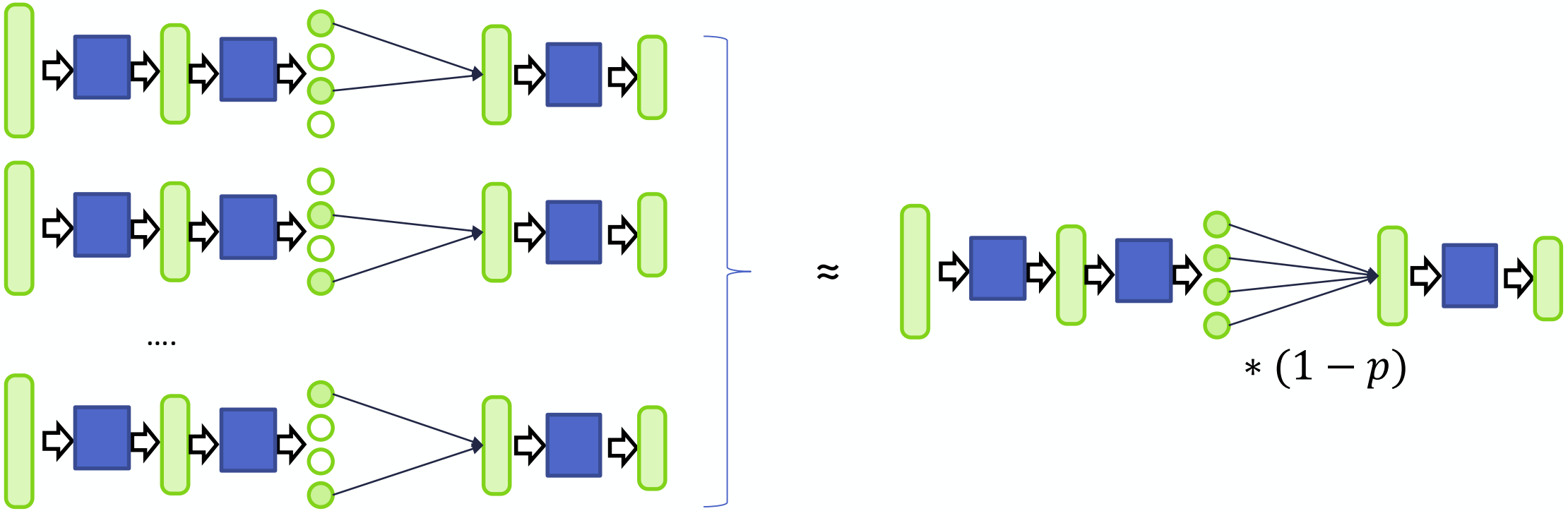
Пусть мы делаем dropout для слоя исходной сети, состоящего из H нейронов.

$1 - p$ — вероятность того, что нейрон включен.

$(1 - p)H$ — среднее количество включенных нейронов.

$C_H^{(1-p)H}$ — общее количество разных сетей, которое можем получить таким образом.

Очень похоже на то, что мы рассматриваем $C_H^{(1-p)H}$ новых сетей и потом усредняем результат. Поэтому также уменьшается переобучение.



Inverted Dropout

Делаем масштабирование не во время тестирования, а во время обучения.

Обучение: $\hat{u}_h = \frac{1}{1-p} X_h u_h$

Тестирование: $\hat{u}_h = u_h$

Во многих фреймворках реализован именно этот вид dropout-а.

- Позволяет не изменять код предсказания.
- Не требует дополнительных операций при тестировании,
  что делает предсказание более быстрым.

ВСЁ!