

НЕЙРОННЫЕ СЕТИ В МАШИННОМ ОБУЧЕНИИ

Лекция №2



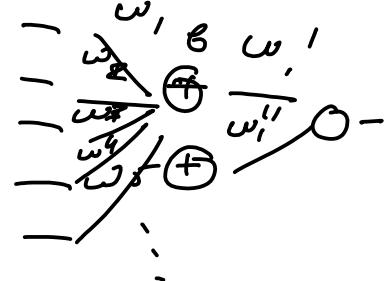
Коммуникация



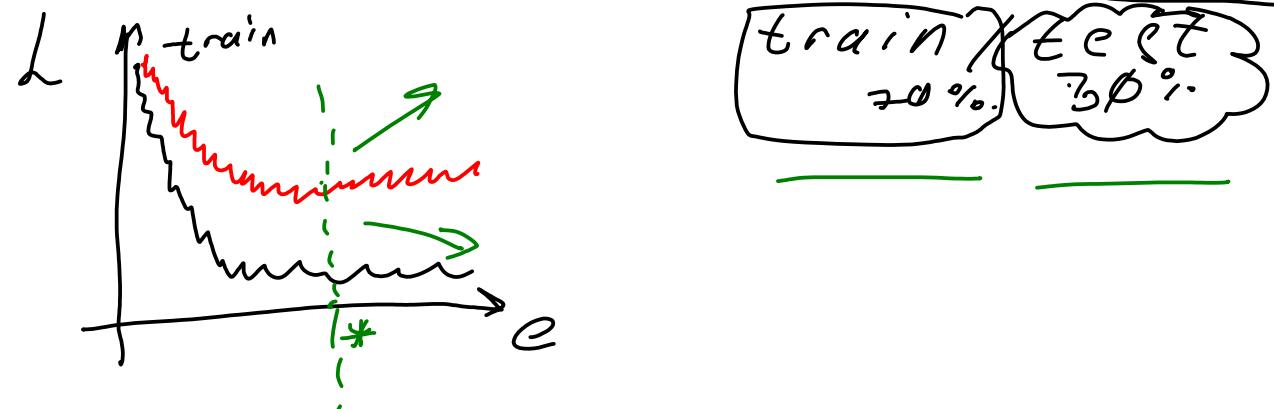
Не забываем отмечаться и оставлять отзывы!

Рекомендации к ДЗ №1

$$y = \underline{w x + b}$$



1. Начинайте реализацию с обучения модели линейной регрессии (для подбора параметров модели используем метод градиентный спуска)
2. Используйте различные функции активации (например, линейный на выходе и сигмоиды на скрытых слоях)
3. Контролируйте обучение (лосс) на тестовой части выборки на каждой эпохе

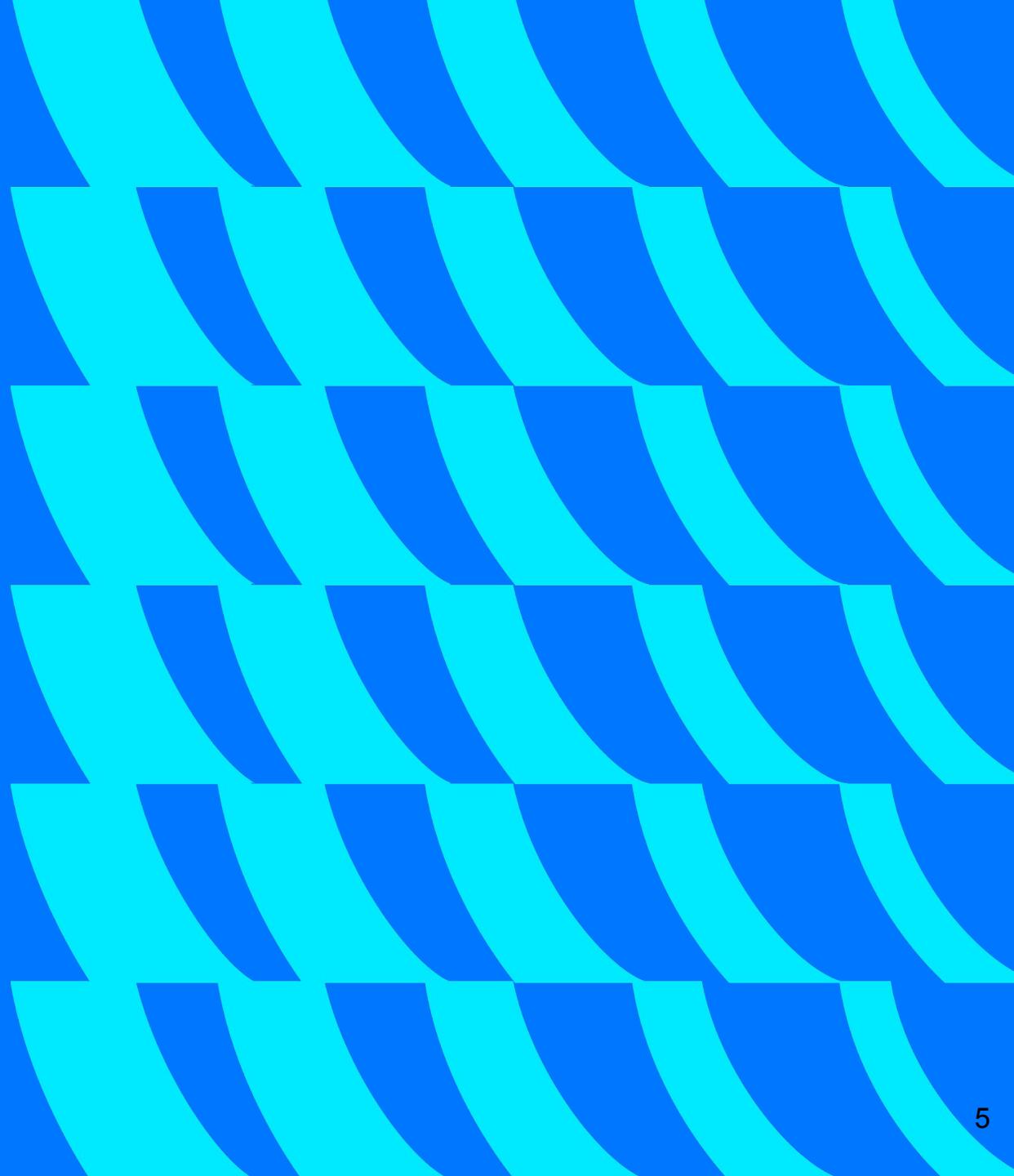


Содержание

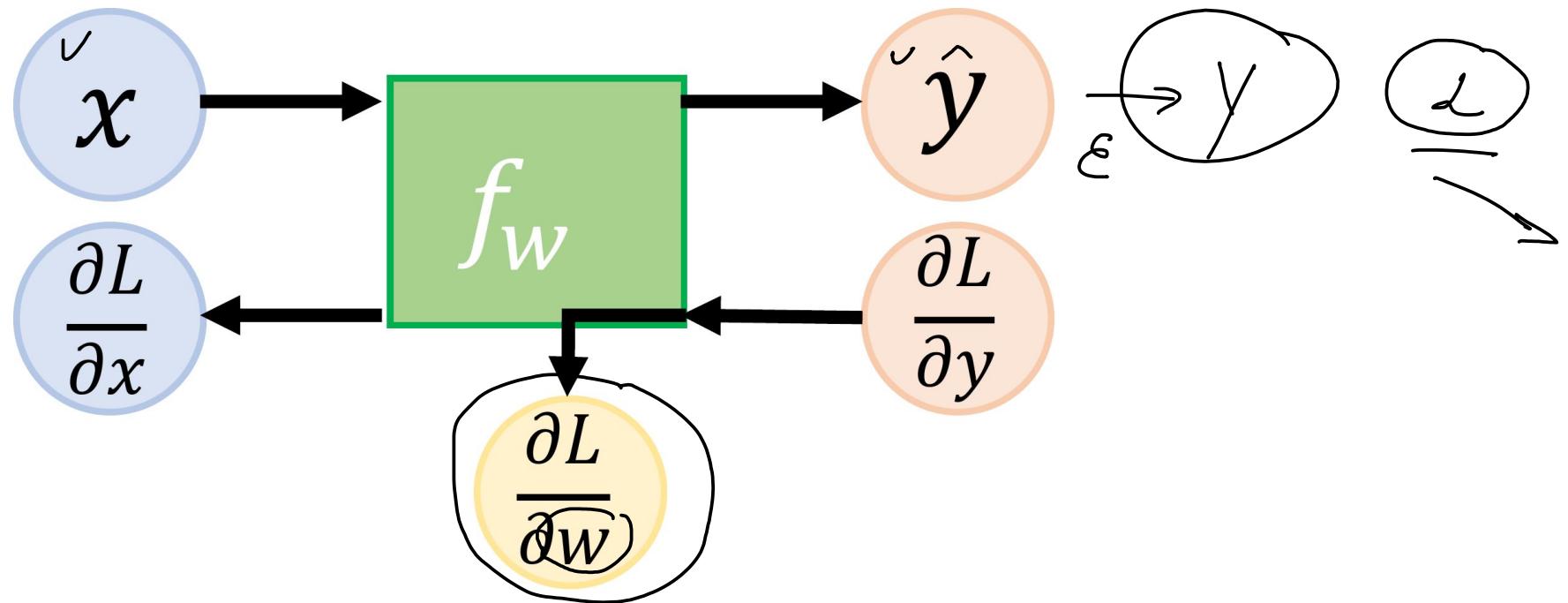
- ✓ 1. Back propagation
- ✓ 2. Ветвящиеся структуры
- ✓ 3. Проблемы обучения нейронных сетей
- ✓ 4. Стохастический градиентный спуск
- 5. PyTorch
- ✓ 6. Домашнее задание

Back propagation

• • • • •



Back propagation



$$\frac{d\mathcal{L}}{dw} = \frac{d\mathcal{L}}{dy} \frac{dy}{dw}$$

$$\frac{d\mathcal{L}}{dx} = \frac{d\mathcal{L}}{dy} \frac{dy}{dx}$$

Back propagation

MLP

$$\text{MSE} = \frac{1}{n} \sum_{i=1}^n (y_i - \hat{y}_i)^2$$

$\leftarrow L(w, w_1, w_2, w_3, w_4, w_5, w_6, b_1, b_2, b_3)$

$w_s = w_s^{t-1} - \frac{\partial L}{\partial w_s}$

$\frac{\partial L}{\partial w_s} = \frac{\partial L}{\partial y_{pred}} \cdot \frac{\partial y_{pred}}{\partial w_s}$

$y_{pred} = f(w_5 \cdot h_1 + w_6 \cdot h_2 + b_3)$

$h_1 = f(x_1 w_1 + x_2 w_2 + b_1)$

$\frac{\partial y_{pred}}{\partial w_s} = \frac{\partial y_{pred}}{\partial h_1} \cdot \frac{\partial h_1}{\partial w_s}$

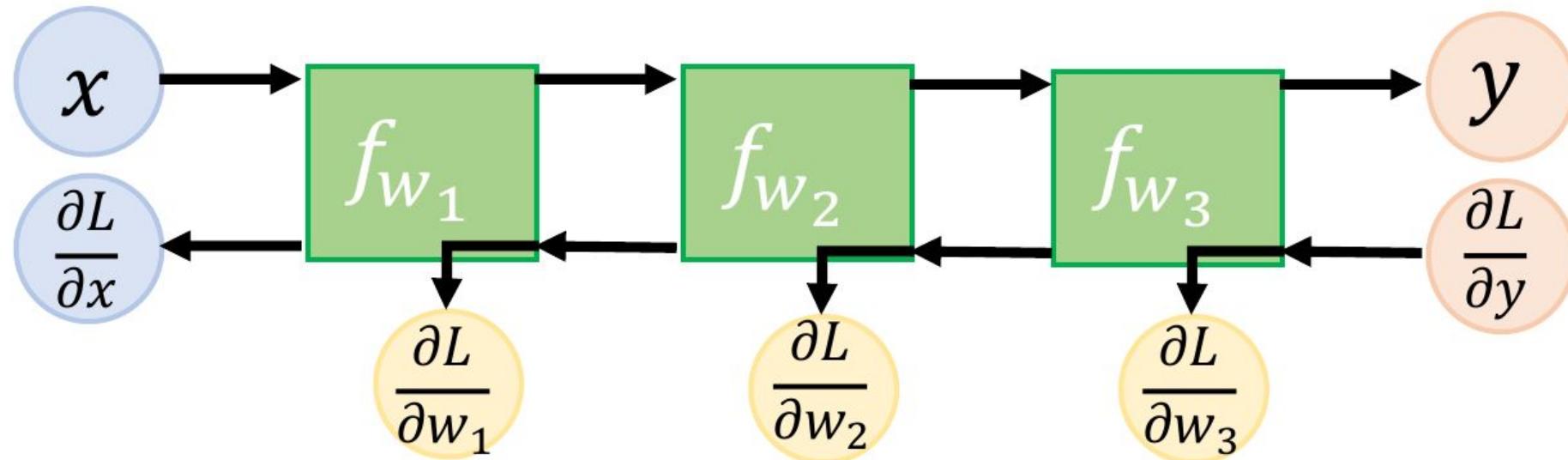
$\frac{\partial h_1}{\partial w_1} = x_1 f'(x_1 w_1 + x_2 w_2 + b_1)$

$f(x) = \frac{1}{1+e^{-x}}$ $f'(x) = \frac{c}{(1+e^{-x})^2} = f(x)(1-f(x))$

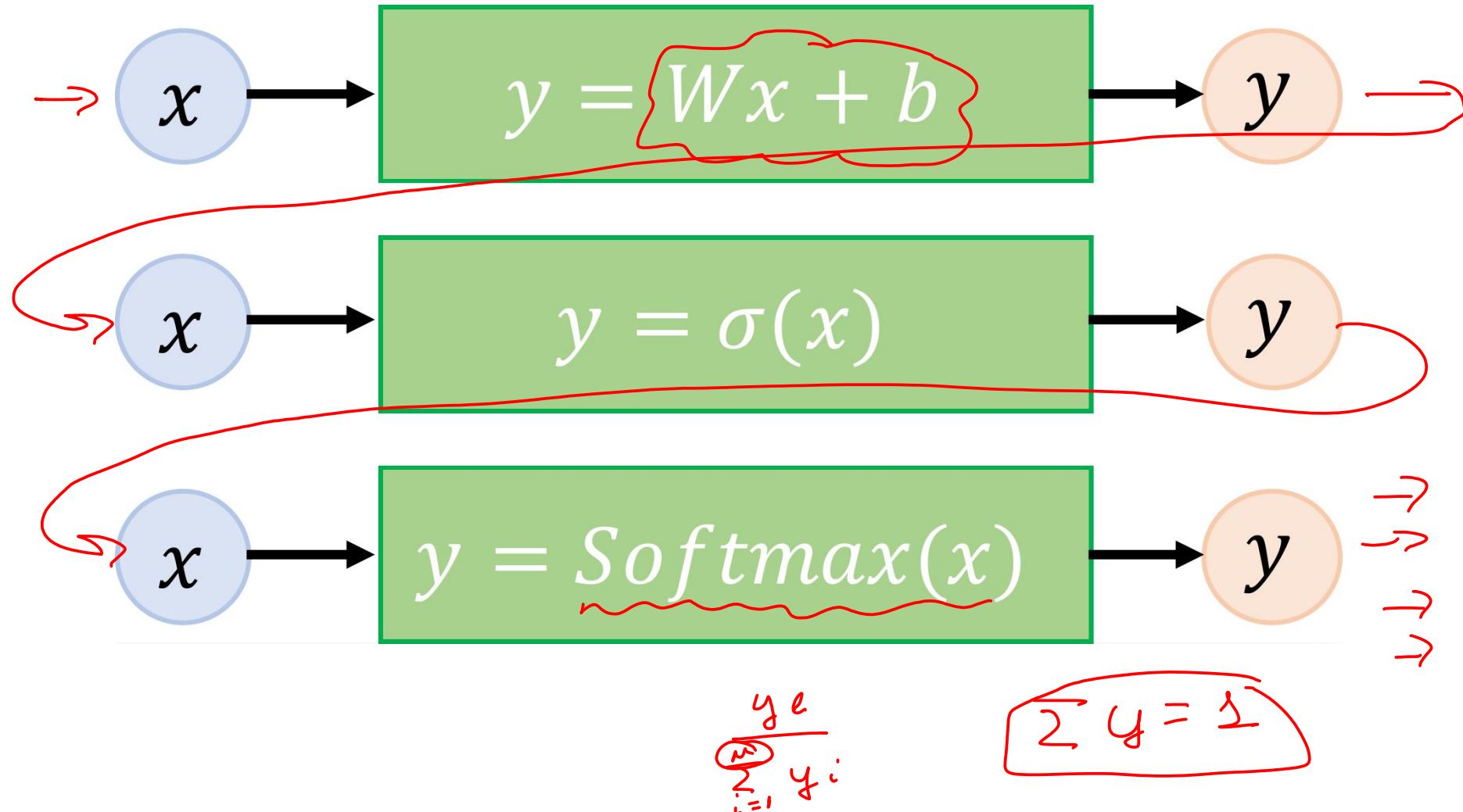
7

Back propagation

Back propagation

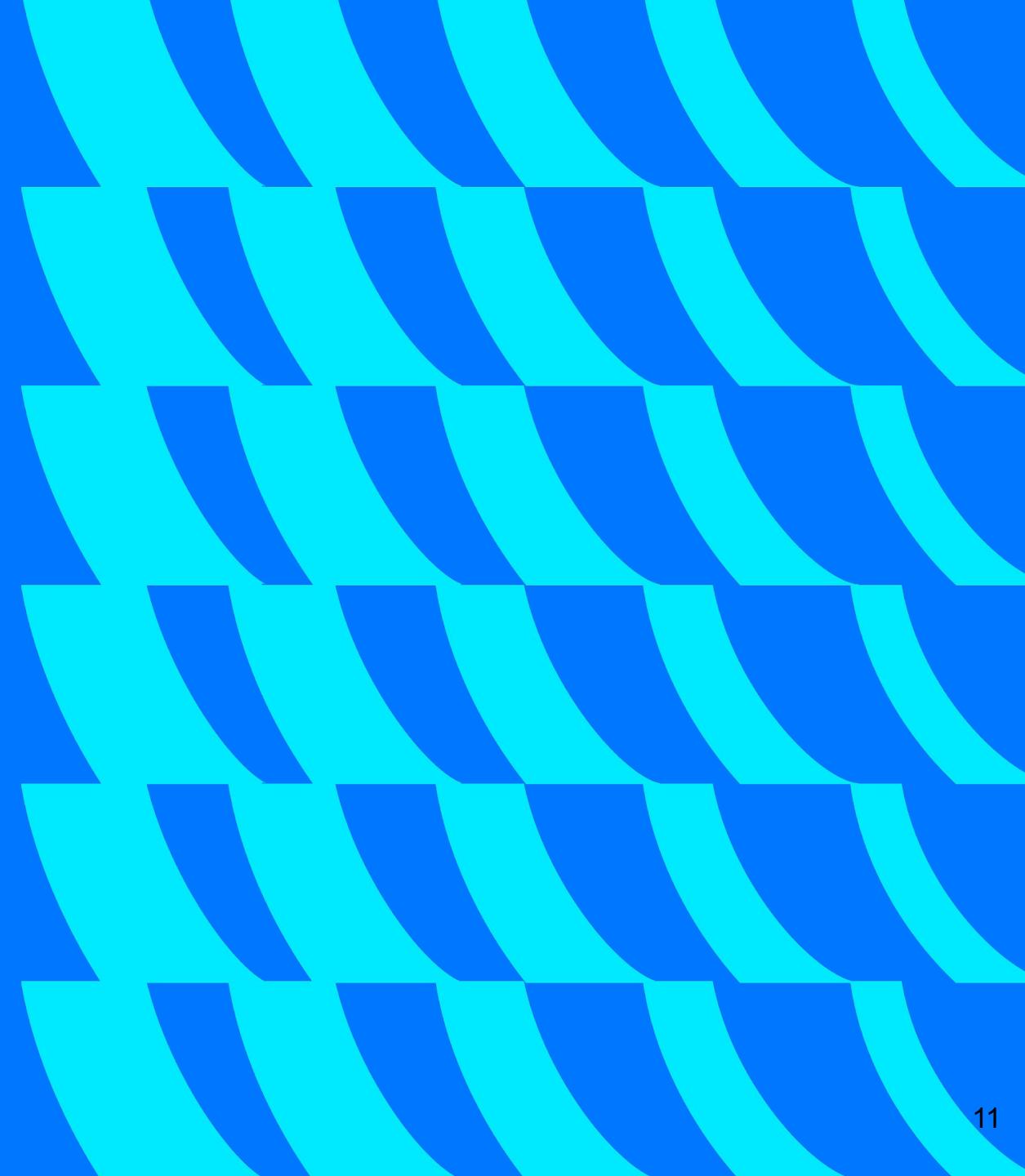


Building blocks

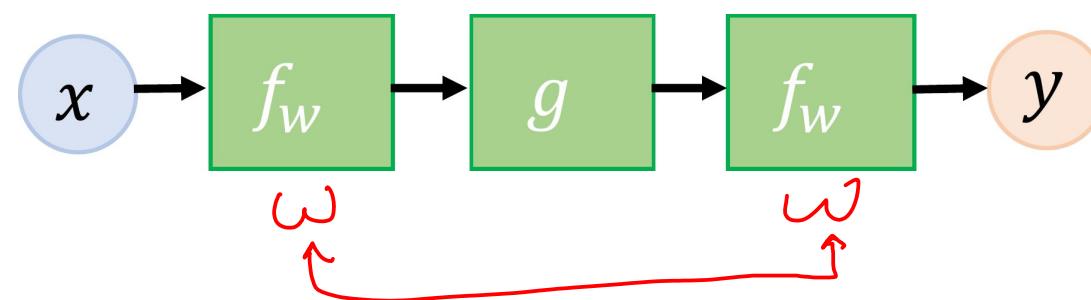
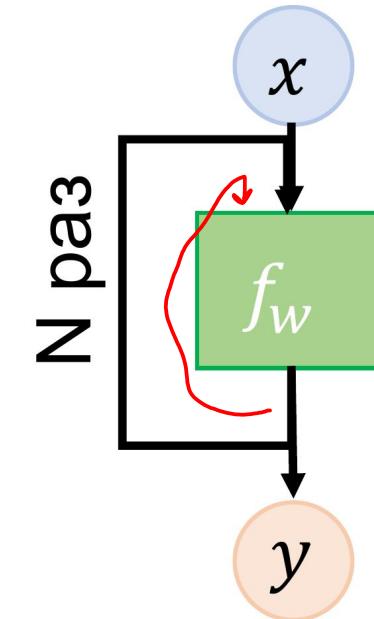
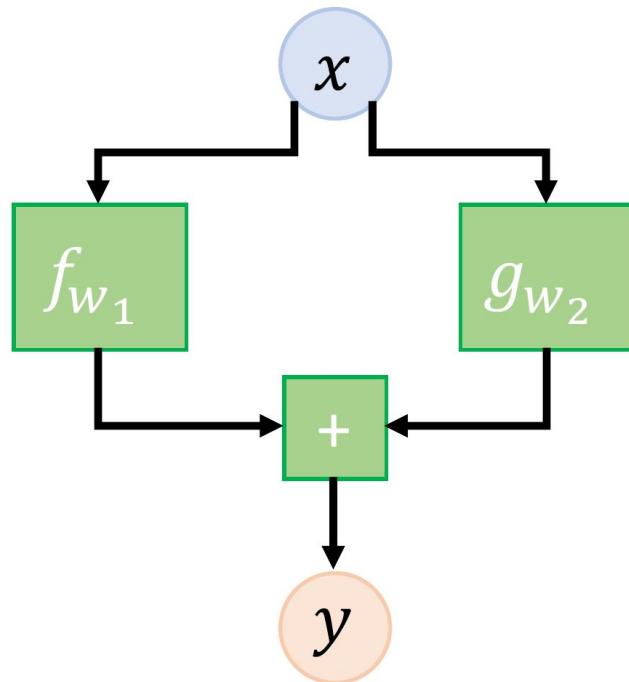


Ветвящиеся структуры

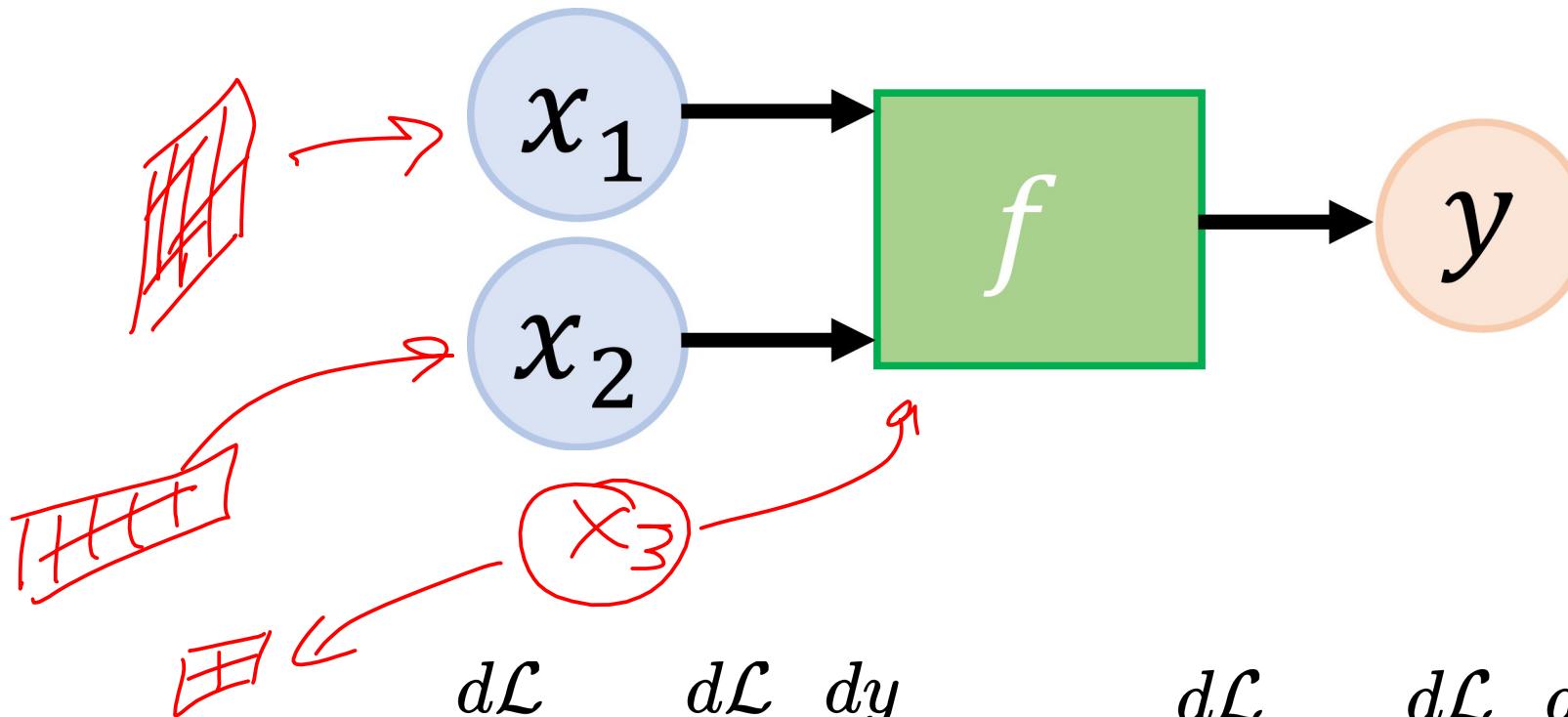
• • • • •



Ветвящиеся структуры



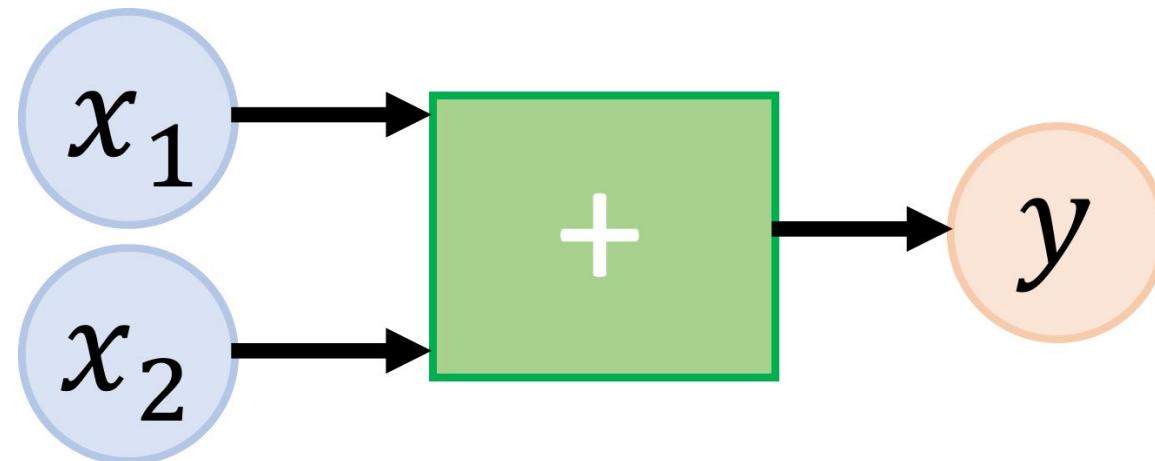
Несколько входов



$$\frac{d\mathcal{L}}{dx_1} = \frac{d\mathcal{L}}{dy} \frac{dy}{dx_1}$$

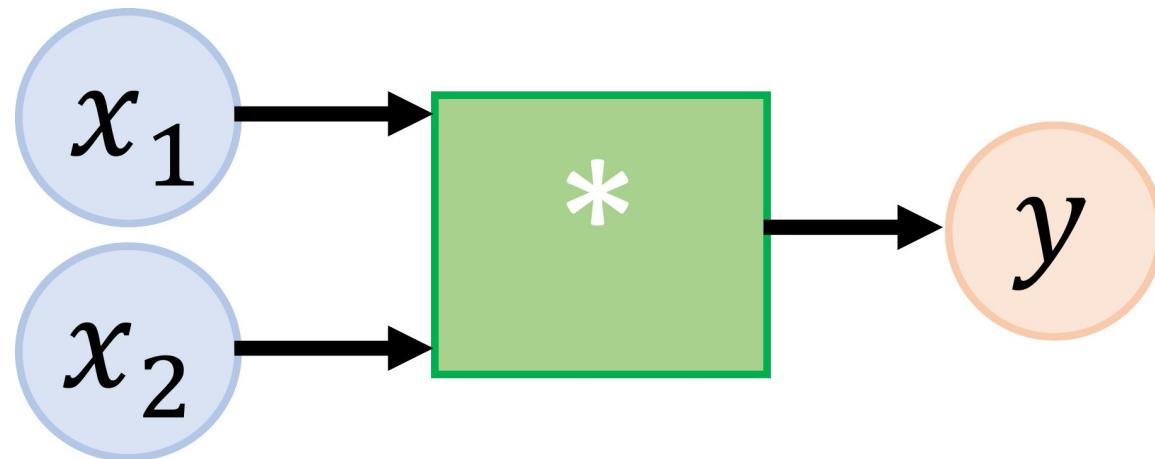
$$\frac{d\mathcal{L}}{dx_2} = \frac{d\mathcal{L}}{dy} \frac{dy}{dx_2}$$

Несколько входов



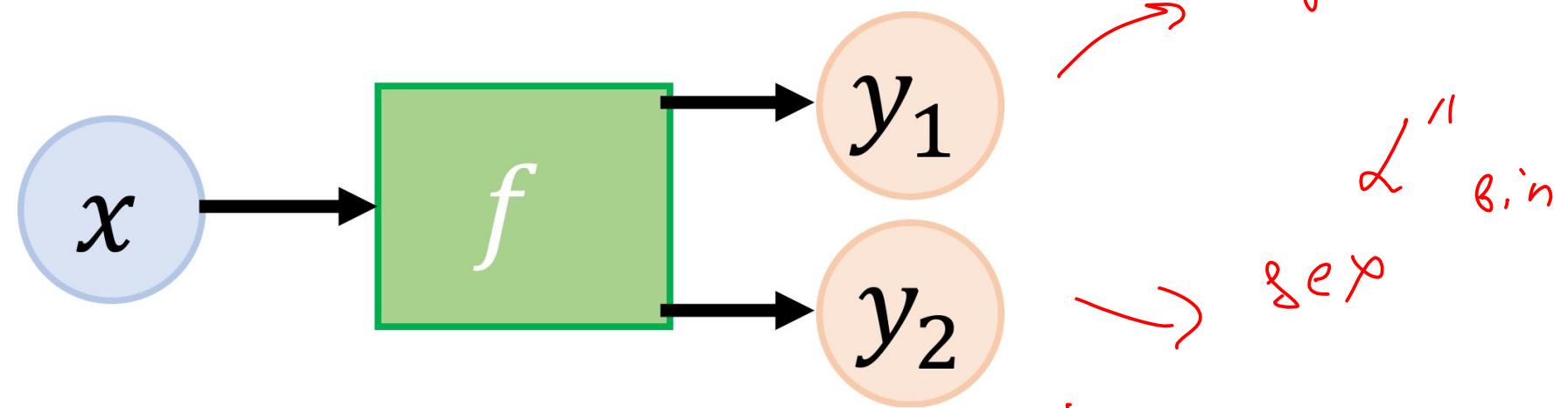
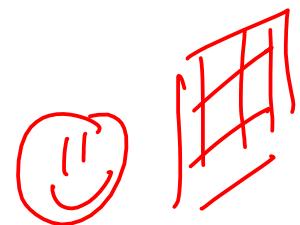
$$\frac{d\mathcal{L}}{dx_1} = \frac{d\mathcal{L}}{dx_2} = \frac{d\mathcal{L}}{dy}$$

Несколько входов



$$\frac{d\mathcal{L}}{dx_1} = \frac{d\mathcal{L}}{dy} x_2, \frac{d\mathcal{L}}{dx_2} = \frac{d\mathcal{L}}{dy} x_1$$

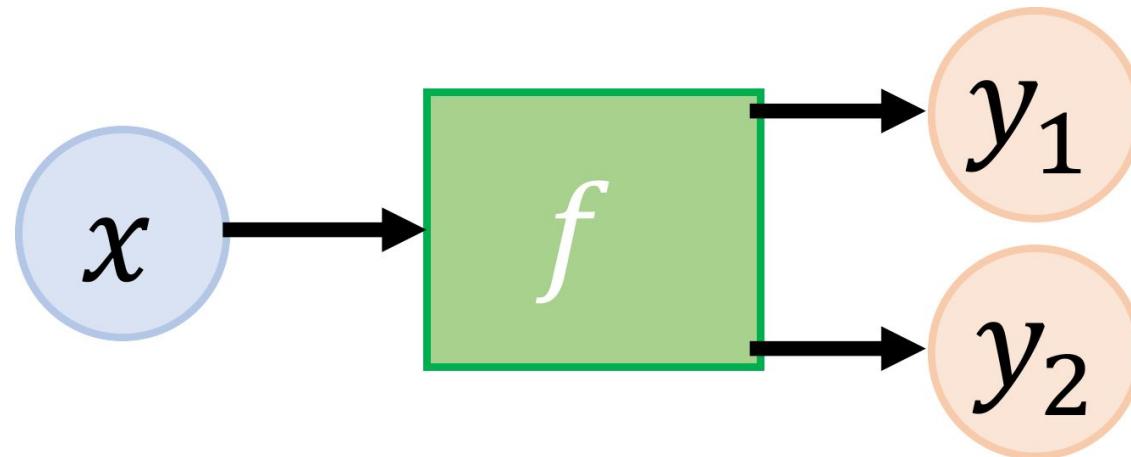
Несколько выходов



$$L = L(y_1(x), y_2(x))$$

$$L = L_{y_1}^{(1)} + L_{y_2}^{(2)} + L_{y_3}^{(3)}$$

Несколько выходов

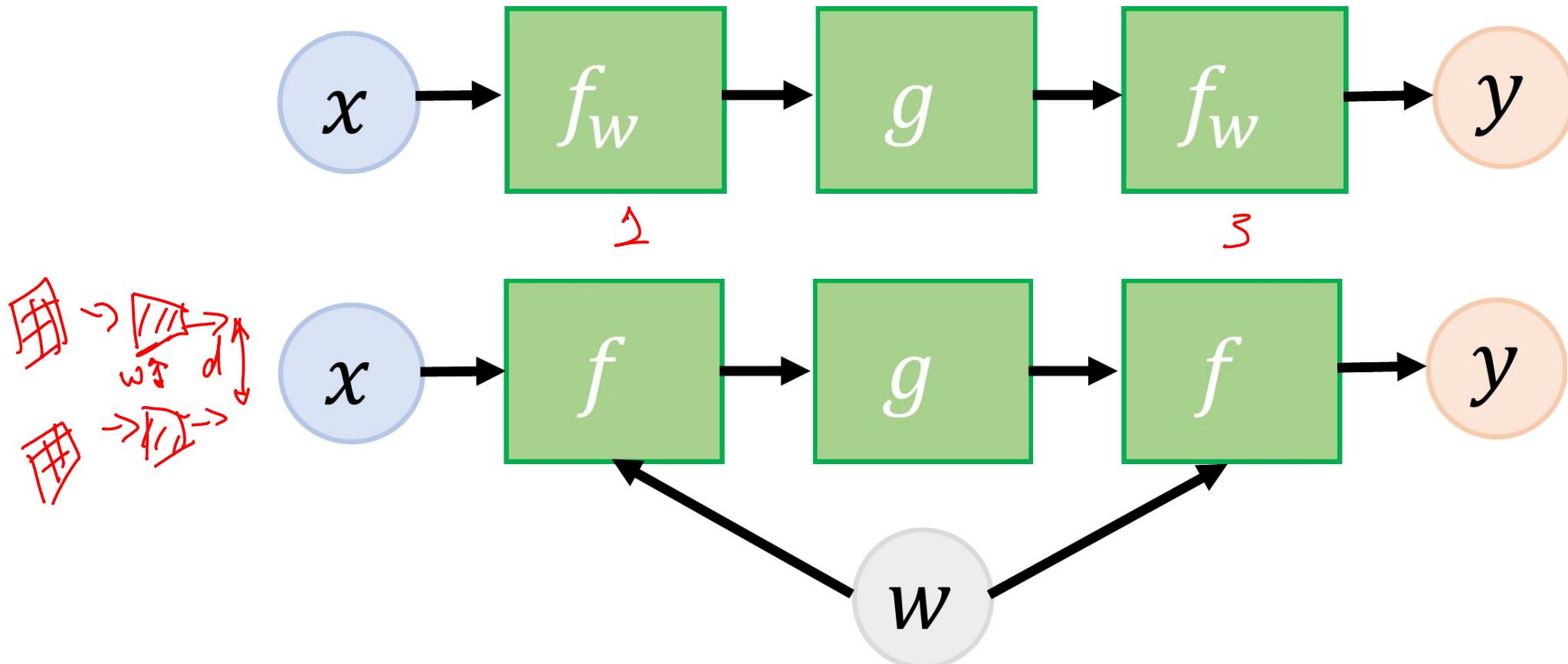


$$L = L(y_1(x), y_2(x)) \implies \frac{d\mathcal{L}}{dx} = \frac{d\mathcal{L}}{dy_1} \frac{dy_1}{dx} + \frac{d\mathcal{L}}{dy_2} \frac{dy_2}{dx}$$

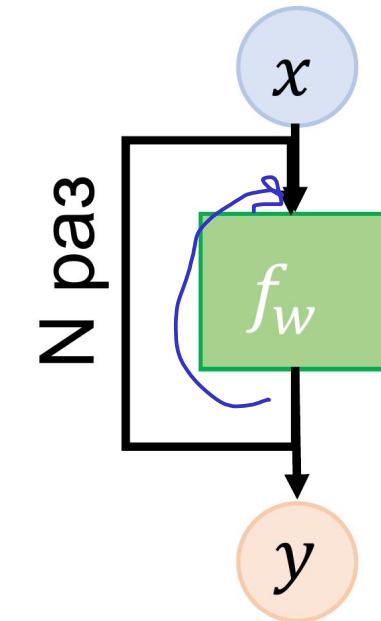
Переиспользование блоков



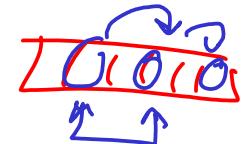
Переиспользование блоков



Рекуррентность



1) текст



2) сигнал

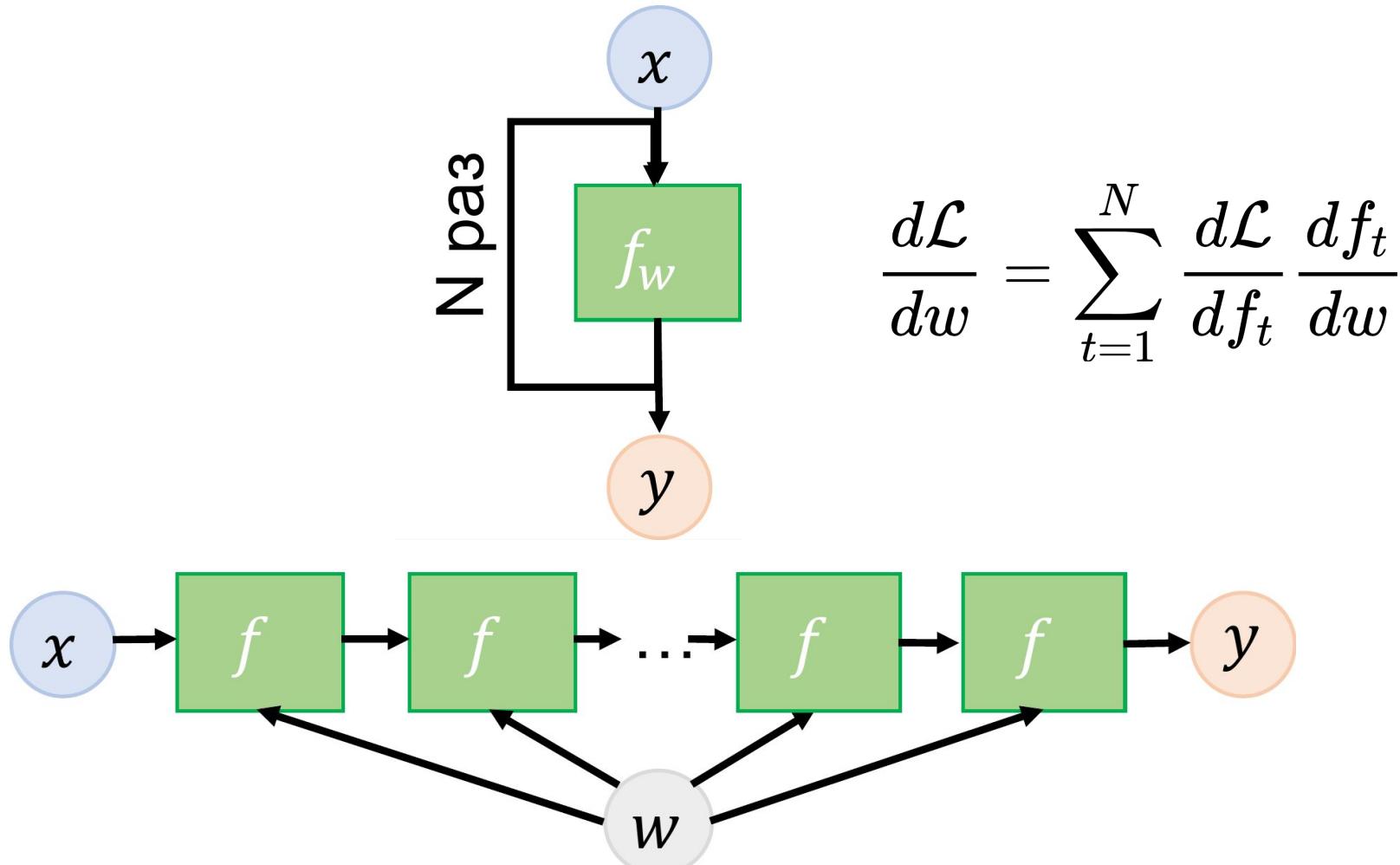


1D

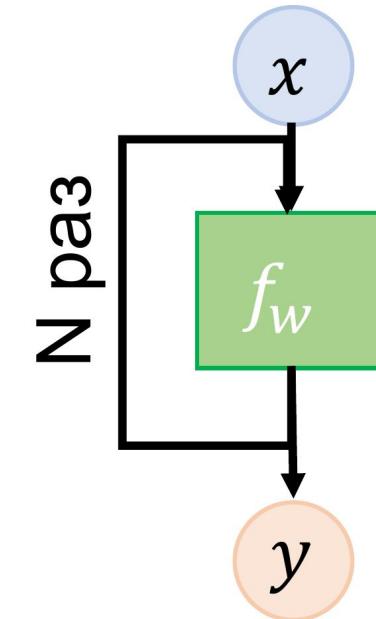
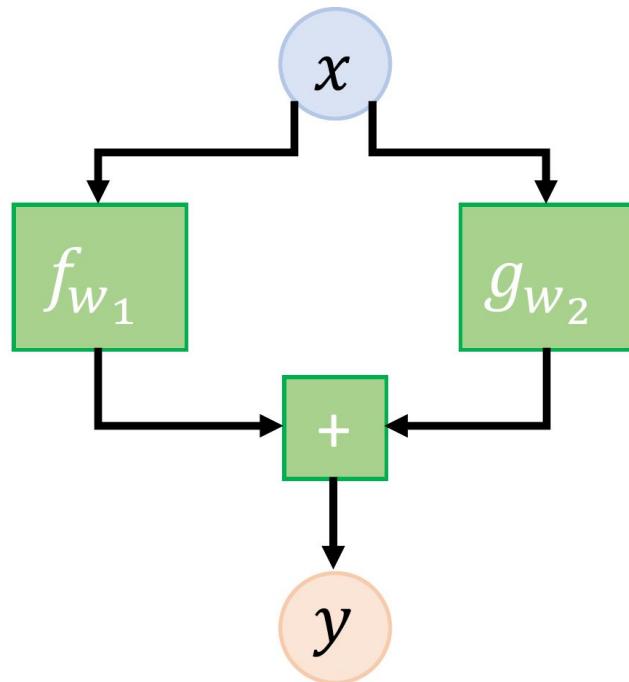
3) 2D



Рекуррентность

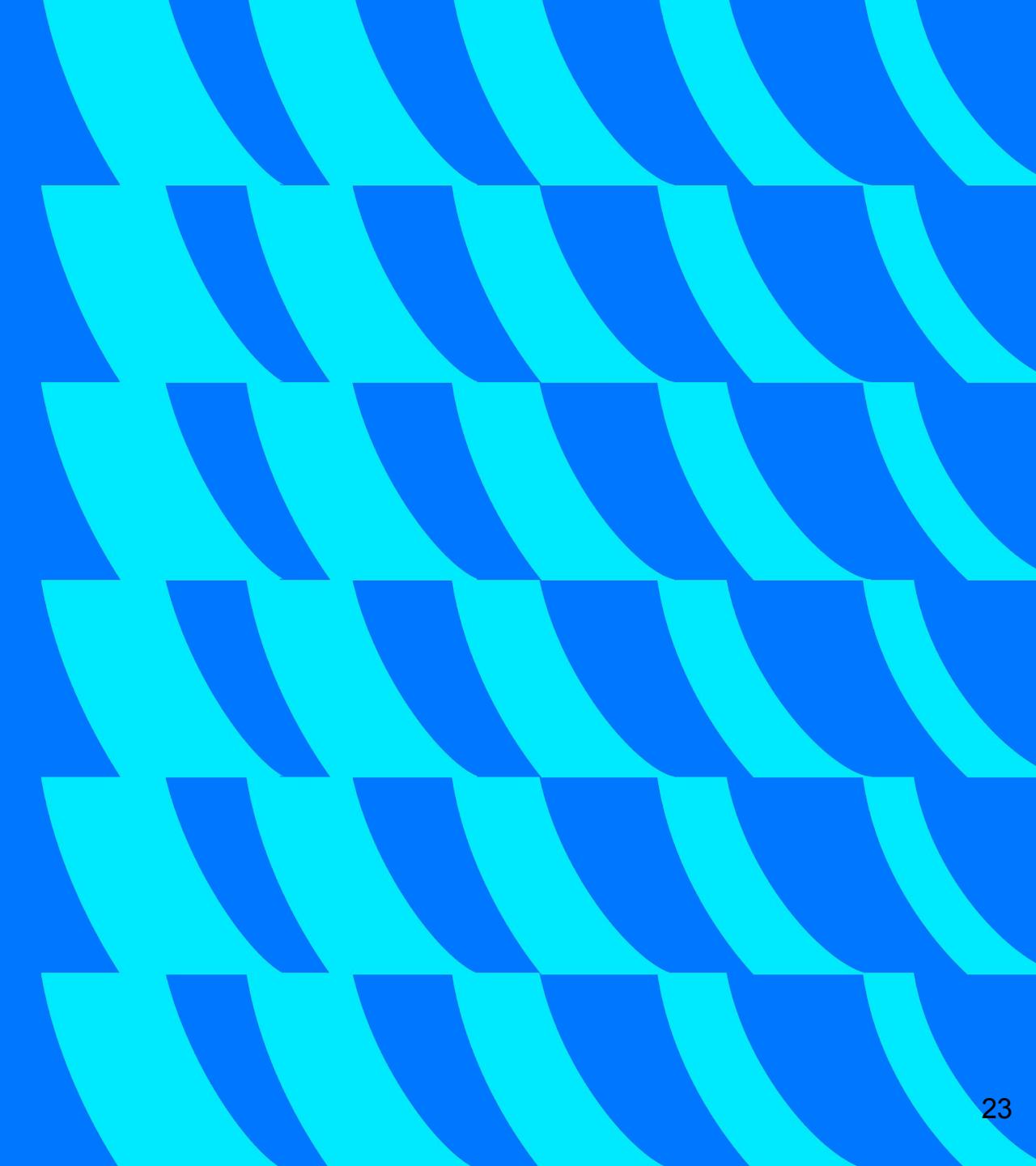


Ветвящиеся структуры



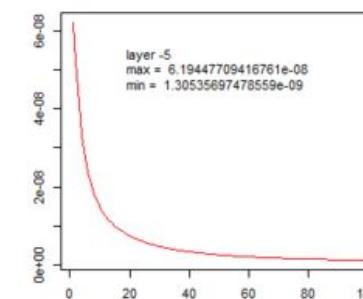
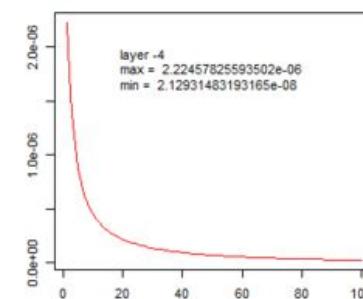
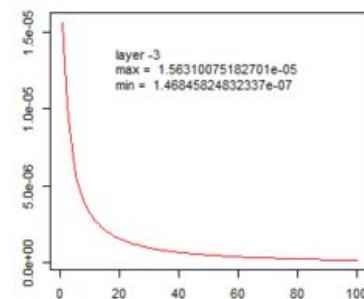
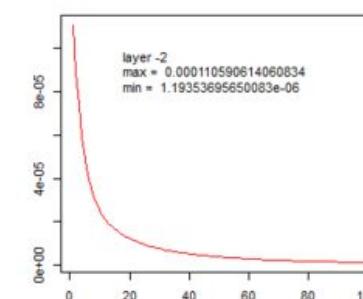
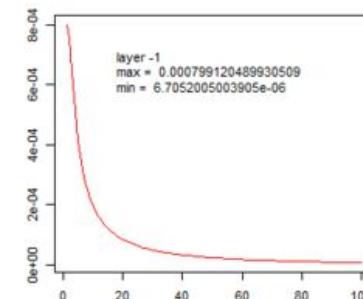
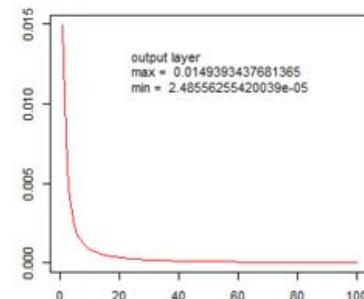
Проблемы обучения нейронных сетей

• • • • •

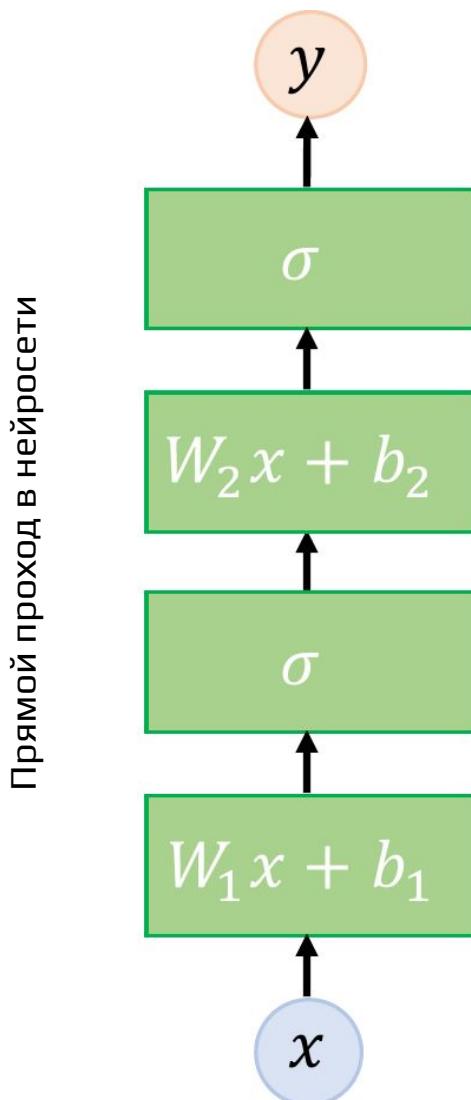


Паралич сети, эксперимент

input [841]	layer -5	layer -4	layer -3	layer -2	layer -1	output
neurons	100	100	100	100	100	26
grad	6.2e-8	2.2e-6	1.6e-5	1.1e-4	7e-4	0.015



Backprop, затухание градиентов



$$y^{(k)} = \sigma\left(x^{(k)} (W^{(k)})^T\right)$$

Выходные значения каждого нейрона σ лежат в интервале $(0, 1)$

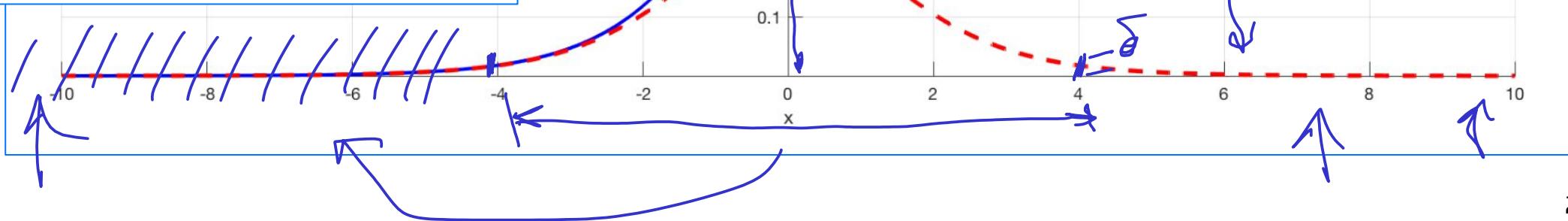
Backprop, затухание градиентов

Рассмотрим в качестве функции активации логистическую функцию:

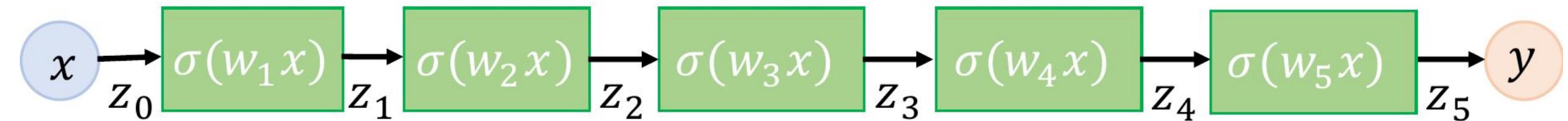
$$\sigma(z) = \frac{1}{1 + e^{-z}}$$

$$\frac{d\sigma}{dz} = \sigma(z) \cdot (1 - \sigma(z))$$

$$\sigma_{\max} = \frac{1}{4}$$
 (0; \frac{1}{4})



Backprop, затухание градиентов



Прямой проход:

$$x = z_0$$

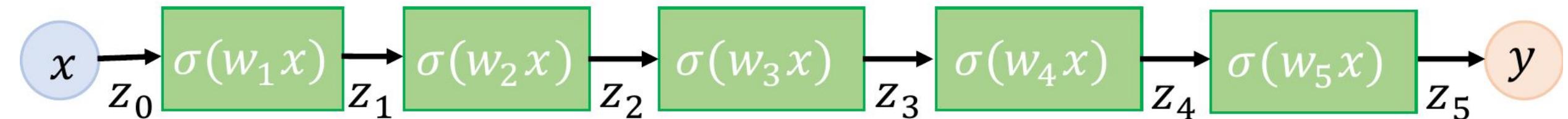
$$z_k = \sigma(z_{k-1} w_k)$$

$$y = z_5$$

Вычислим градиенты весов для: $L(y, t) = \frac{1}{2}(y_j - t_j)^2$

$$\frac{d\mathcal{L}}{dz_4} =$$

Backprop, затухание градиентов



Прямой проход:

$$x = z_0$$

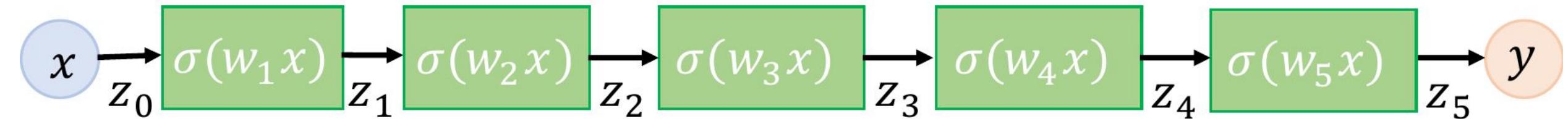
$$z_k = \sigma(z_{k-1} w_k)$$

$$y = z_5$$

Вычислим градиенты весов для: $L(y, t) = \frac{1}{2}(y_j - t_j)^2$ $\max = \frac{1}{4}$

$$\frac{d\mathcal{L}}{dz_4} = \frac{d\mathcal{L}}{dy} \frac{dy}{dz_4} = (y - t) \sigma'(w_5 z_5) w_5 \leq 2 \cdot \underbrace{\frac{1}{4} w_5}_{-}$$

Backprop, затухание градиентов



Прямой проход:

$$x = z_0$$

$$z_k = \sigma(z_{k-1} w_k)$$

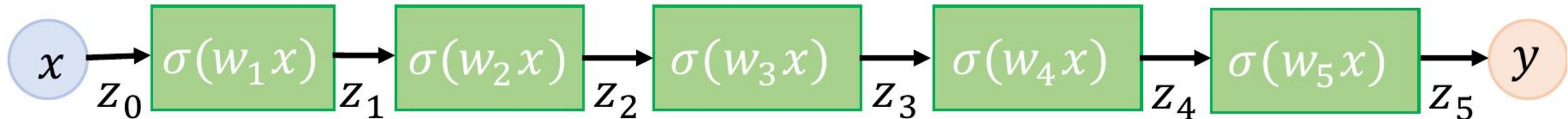
$$y = z_5$$

Вычислим градиенты весов для: $L(y, t) = \frac{1}{2}(y_j - t_j)^2$

$$\frac{d\mathcal{L}}{dz_4} = \frac{d\mathcal{L}}{dy} \frac{dy}{dz_4} = (y - t) \sigma'(w_5 z_5) w_5 \leq 2 \cdot \frac{1}{4} w_5$$

$$\frac{d\mathcal{L}}{dz_3} = \frac{d\mathcal{L}}{dz_4} \frac{dz_4}{dz_3} \leq 2 \cdot \left(\frac{1}{4}\right)^2 w_4 w_5$$

Backprop, затухание градиентов



Прямой проход:

$$x = z_0$$

$$z_k = \sigma(z_{k-1} w_k)$$

$$y = z_5$$

Вычислим градиенты весов для: $L(y, t) = \frac{1}{2}(y_j - t_j)^2$

$$\frac{d\mathcal{L}}{dz_4} = \frac{d\mathcal{L}}{dy} \frac{dy}{dz_4} = (y - t) \sigma'(w_5 z_5) w_5 \leq 2 \cdot \frac{1}{4} w_5 \quad \leftarrow$$

$$\frac{d\mathcal{L}}{dz_3} = \frac{d\mathcal{L}}{dz_4} \frac{dz_4}{dz_3} \leq 2 \cdot \left(\frac{1}{4}\right)^2 w_4 w_5$$

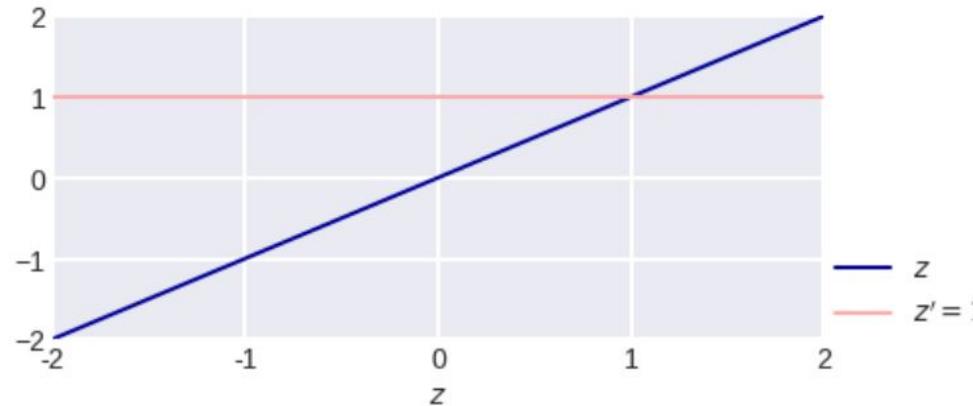
$$\frac{d\mathcal{L}}{dx} = \frac{d\mathcal{L}}{dz_1} \frac{dz_1}{dx} \leq 2 \cdot \left(\frac{1}{4}\right)^5 w_1 w_2 w_3 w_4 w_5 \quad \leftarrow \left(\frac{1}{4}\right)^6 \left(\frac{1}{4}\right)^7 \dots$$

Backprop, затухание градиентов, выводы

1. Значение градиента затухает экспоненциально \Rightarrow сходимость замедляется
2. При малых значениях весов этот эффект усиливается
3. При больших значениях весов значение градиента может экспоненциально возрастать \Rightarrow алгоритм расходится
4. Эффект мало заметен у сетей с малым числом слоев

ФУНКЦИИ АКТИВАЦИИ

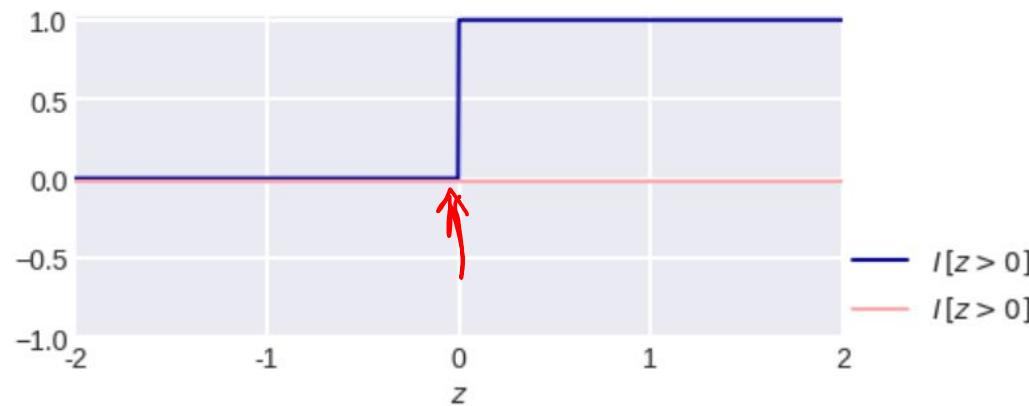
Тождественная функция (линейная / linear activation function)



$$f(z) = z$$

$$\frac{\partial f(z)}{\partial z} = 1$$

Пороговая функция (threshold function)

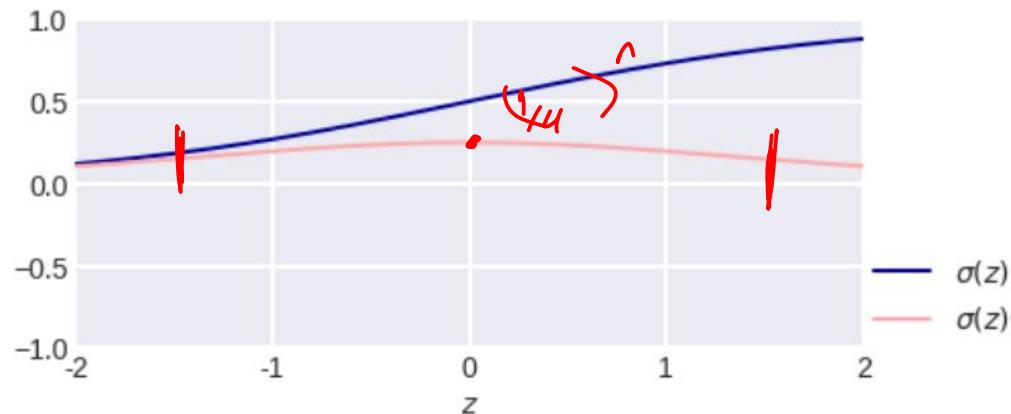


$$\text{th}(z) = I[z > 0]$$

$$\frac{\partial \text{th}(z)}{\partial z} = 0$$

ФУНКЦИИ АКТИВАЦИИ

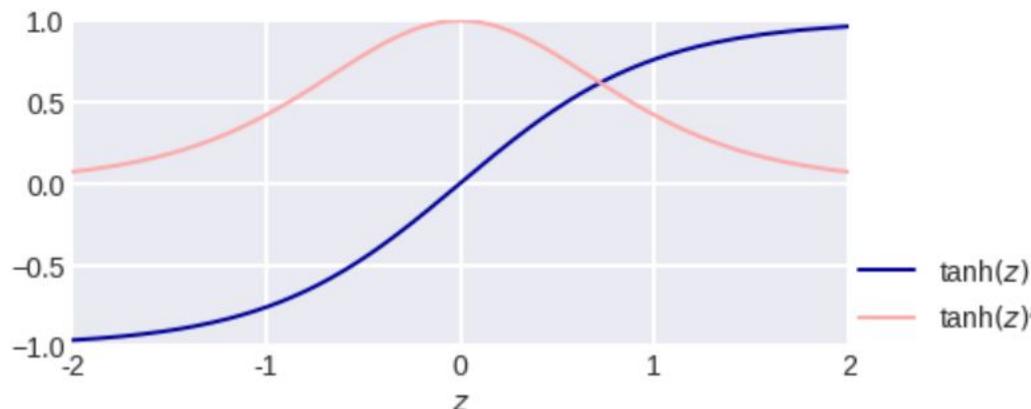
Сигмоида (sigmoid activation function)



$$\sigma(z) = \frac{1}{1 + e^{-z}} \in (0, 1)$$

$$\frac{\partial \sigma(z)}{\partial z} = \sigma(z)(1 - \sigma(z)) > 0$$

Гиперболический тангенс (hyperbolic tangent)



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

$$\frac{\partial \tanh(z)}{\partial z} = 1 - \tanh^2(z)$$

ФУНКЦИИ АКТИВАЦИИ

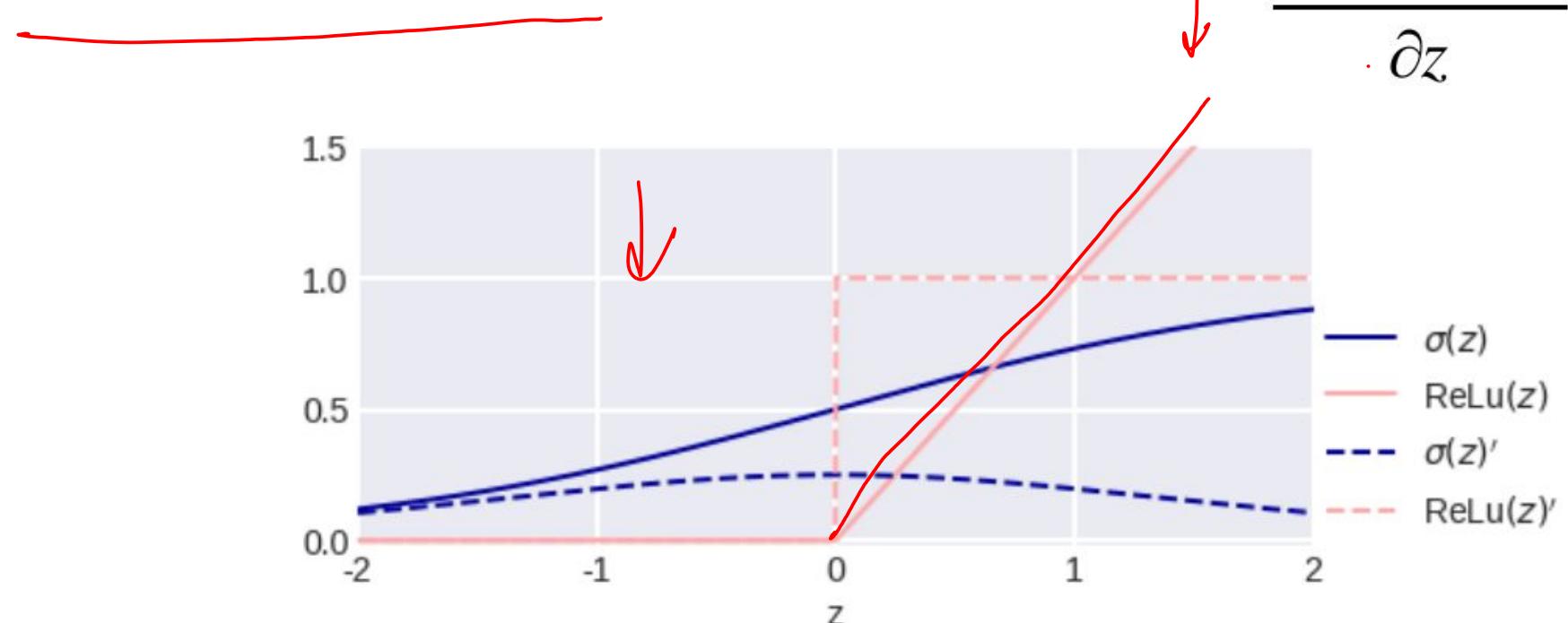
ReLU = Rectified Linear Unit

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

$$\frac{\partial \sigma(z)}{\partial z} = \sigma(z)(1 - \sigma(z))$$

$$\text{ReLU}(z) = \max(0, z)$$

$$\frac{\partial \text{ReLU}(z)}{\partial z} = I[z > 0]$$



Функции активации

Что плохого в сигмоиде

1. «убивает» градиенты
2. выходы не отцентрированы (легко устранить \tanh)
3. вычисление экспоненты всё-таки дорого...

Что хорошего в ReLU

1. быстро вычисляется
2. есть теоретические / биологические обоснования
3. зоны константного градиента и обнуления (разреженное решение)

ФУНКЦИИ АКТИВАЦИИ

$$\text{LeakyReLU}(z) = \max(0.1z, z)$$

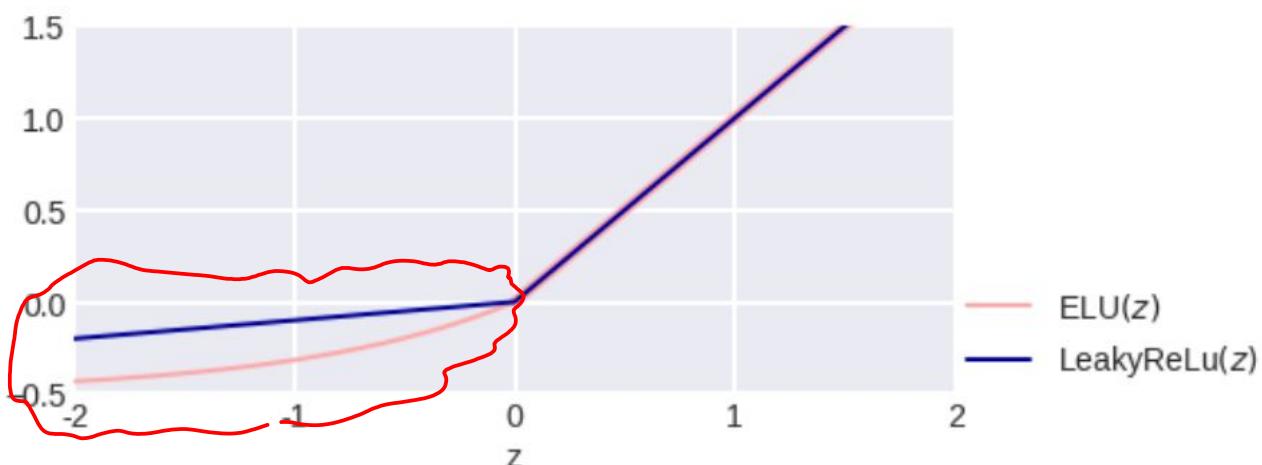
—

Exponential Linear Unit

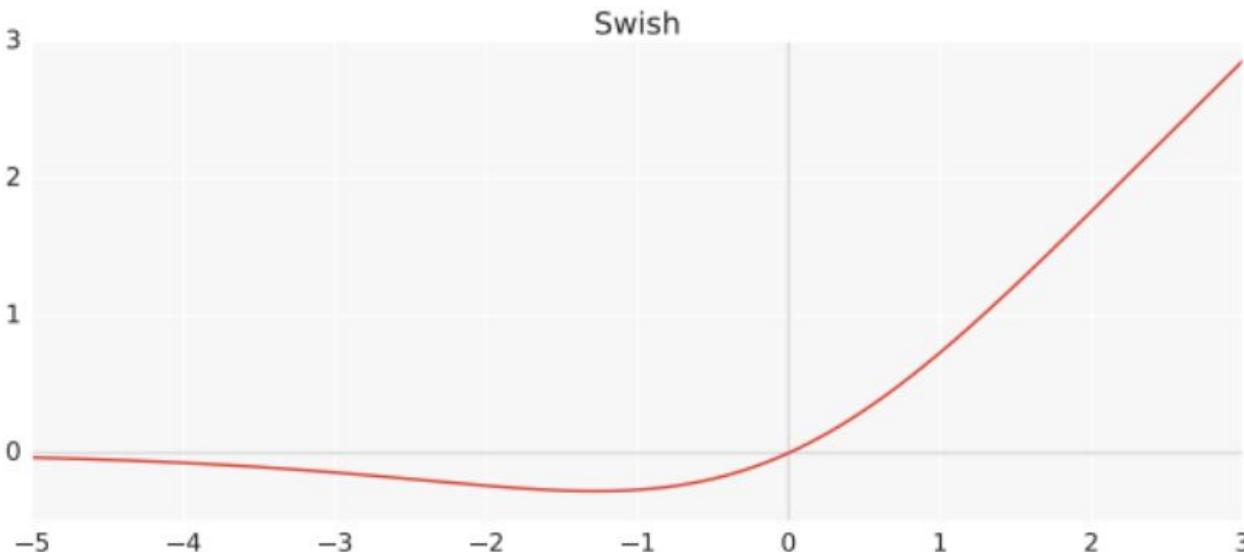
$$\text{ELU}(z) = \begin{cases} z, & z \geq 0, \\ \alpha(e^z - 1), & z < 0. \end{cases}$$

Scaled Exponential Linear Unit

$$\text{SELU}(z) = \lambda \text{ELU}(z)$$



ФУНКЦИИ АКТИВАЦИИ



SILU

CLASS `torch.nn.SiLU(inplace=False)`

Swish

$$\text{swish}(z) = x \cdot \sigma(\alpha x)$$

При замене ReLu на swish в глубоких сетях немного (<1%) улучшается качество

Applies the Sigmoid Linear Unit (SiLU) function, element-wise. The SiLU function is also known as the swish function.

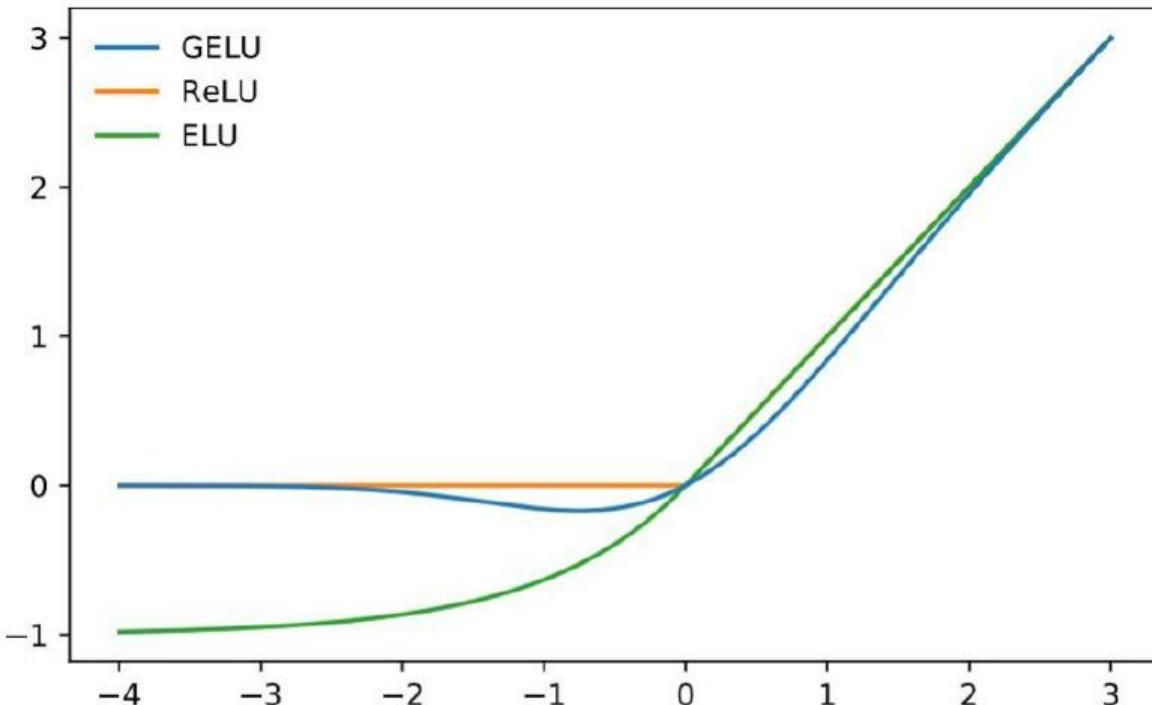
$$\text{silu}(x) = x * \sigma(x), \text{ where } \sigma(x) \text{ is the logistic sigmoid.}$$

• NOTE

See [Gaussian Error Linear Units \(GELUs\)](#) where the SiLU (Sigmoid Linear Unit) was originally coined, and see [Sigmoid-Weighted Linear Units for Neural Network Function Approximation in Reinforcement Learning](#) and [Swish: a Self-Gated Activation Function](#) where the SiLU was experimented with later.

ФУНКЦИИ АКТИВАЦИИ

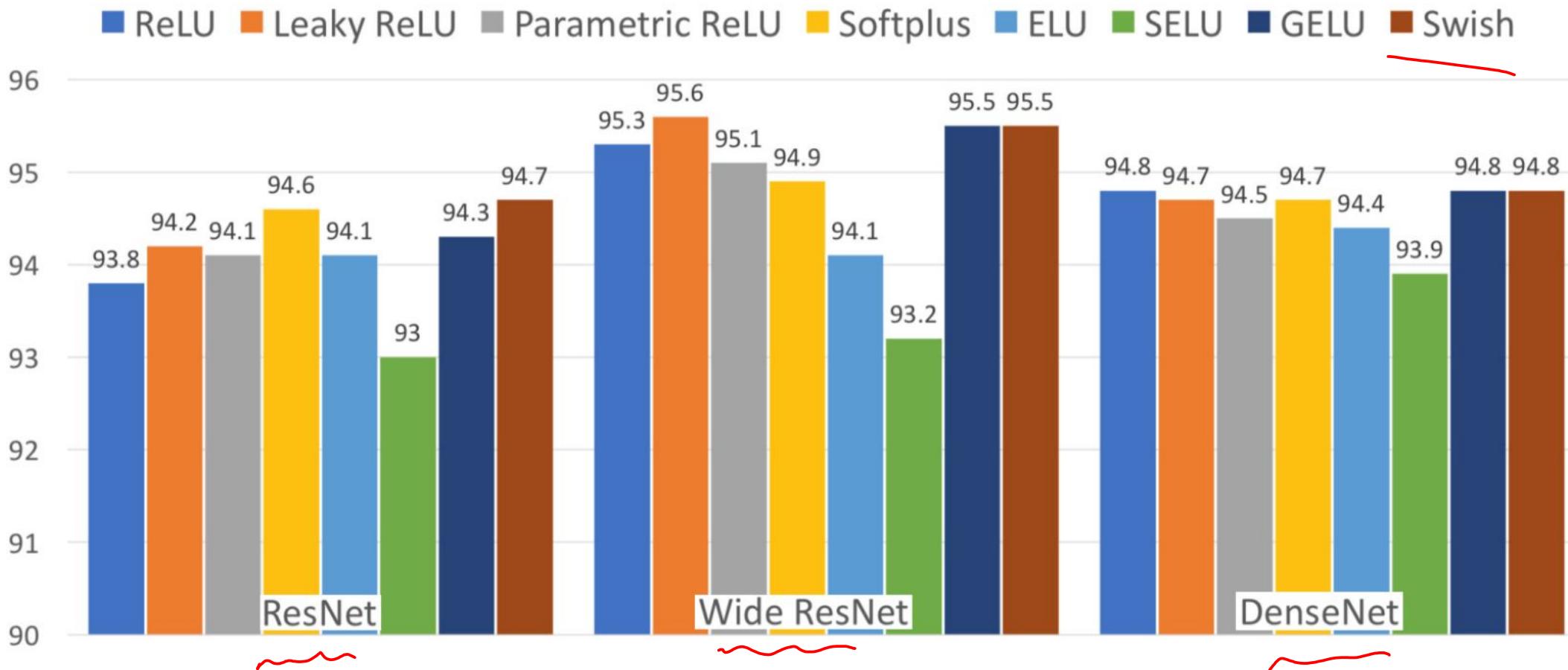
Gaussian Error Linear Unit (Google's BERT, OpenAI's GPT-2)



$$\text{GELU}(z) = \frac{z}{2} \left(1 + \tanh \left(\sqrt{\frac{2}{\pi}} (z + \alpha z^3) \right) \right)$$
$$\alpha = 0.044715$$

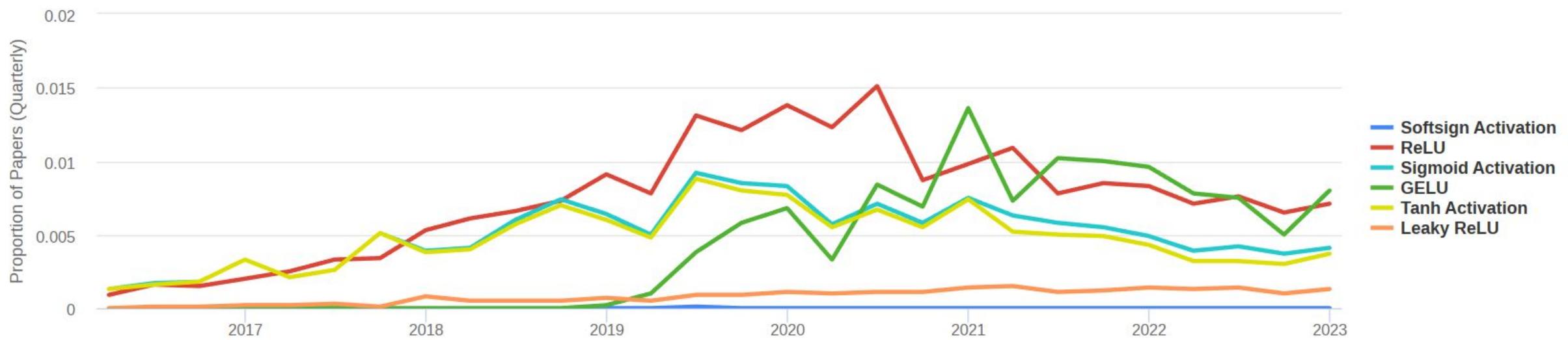
Figure 1: The GELU ($\mu = 0, \sigma = 1$), ReLU, and ELU ($\alpha = 1$).

ФУНКЦИИ АКТИВАЦИИ



<https://arxiv.org/pdf/1710.05941.pdf>

ФУНКЦИИ АКТИВАЦИИ



<https://paperswithcode.com/method/softsign-activation>

Переобучение: Аугментация

Искусственно увеличиваем выборку:

1. Небольшие вращения
2. Небольшие отражения
3. Небольшие изменения в цвете

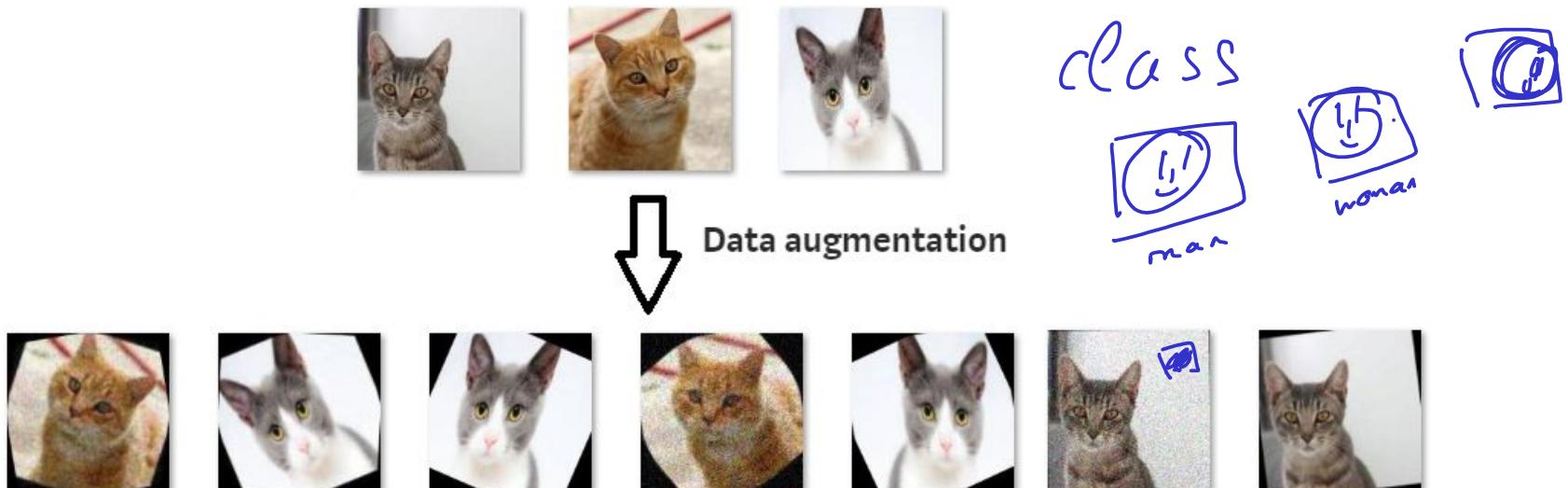
4. Небольшие сдвиги

5. ... noise

mix

mask

⋮
⋮
⋮
⋮



Переобучение: Регуляризация

Дополнительный штраф: $\overbrace{L_R = L(\vec{y}, \vec{t}) + \lambda \cdot R(W)}$

L2 регуляризация: Помогает бороться с мультиколлинеарностью

$$R_{L2}(W) = \frac{1}{2} \sum_i w_i^2$$

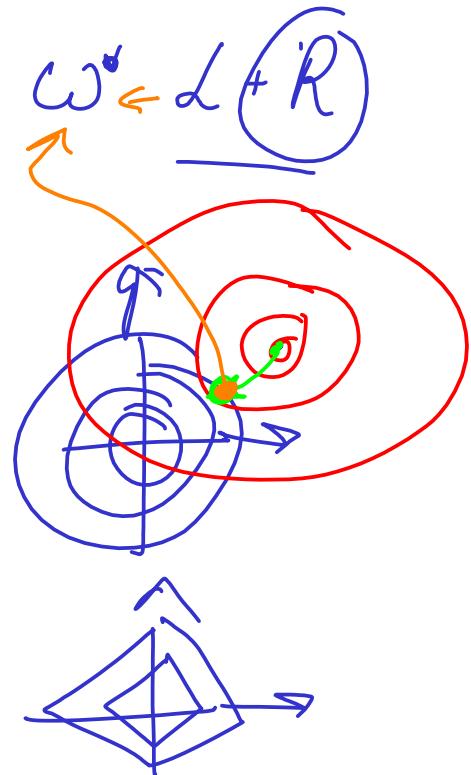
$$\overbrace{L_1 + L_2}$$

$$\frac{dR_{L2}(W)}{dw_i} = \overbrace{w_i w_i^2}$$

L1 регуляризация: Поощряет разреженные веса

$$R_{L1}(W) = \sum_i |w_i|$$

$$\frac{dR_{L1}(W)}{dw_i} = \text{sign}(w_i)$$



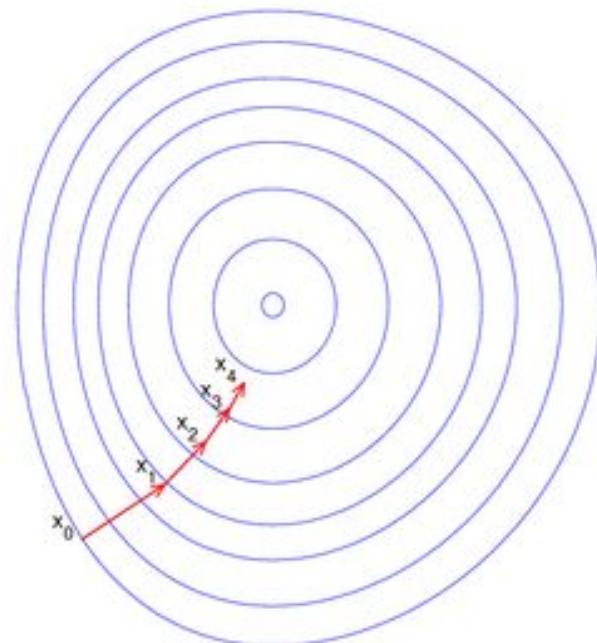
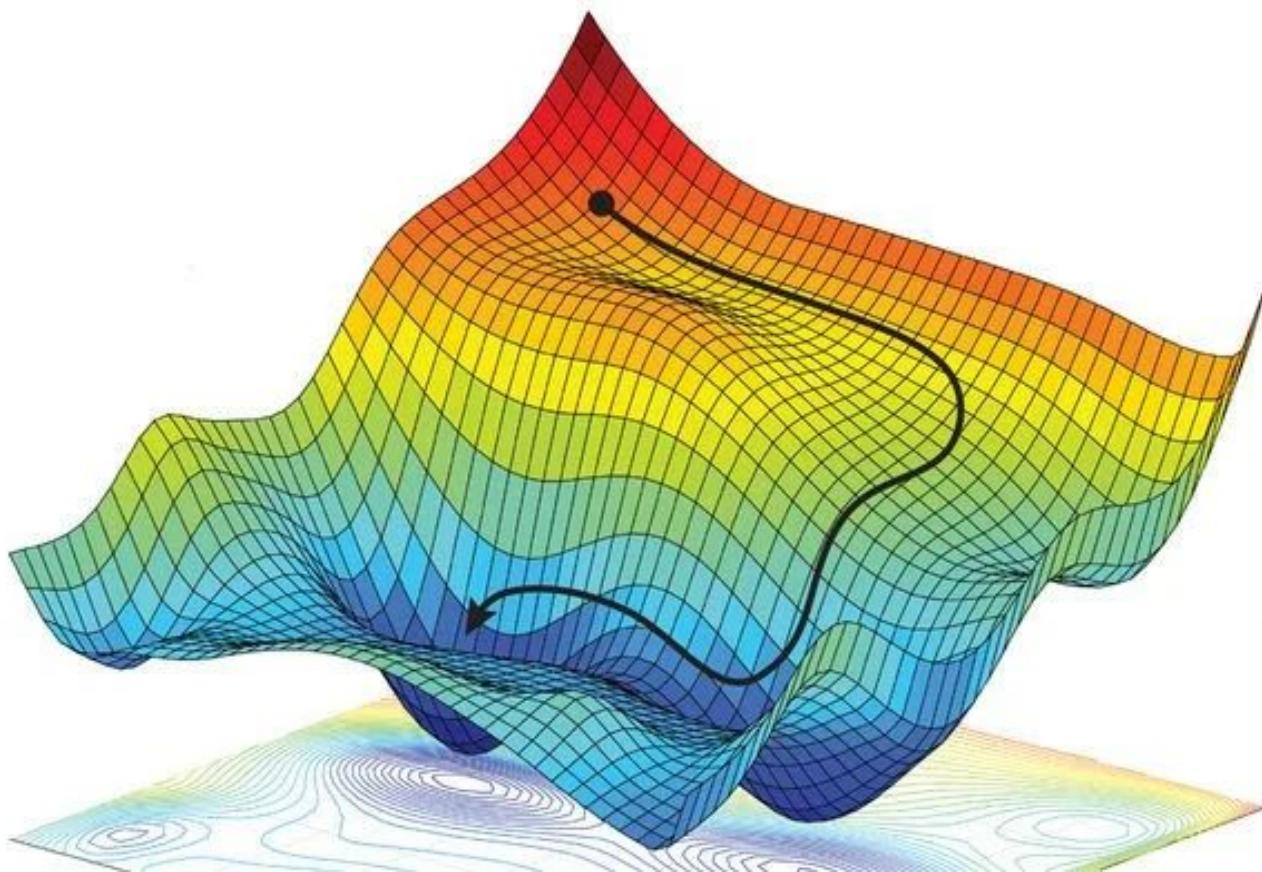
Стохастический градиентный спуск

• • • • •



Постановка задачи

1. $\theta_* = \min_{\theta} J(\theta)$
2. В любой точке можем вычислить $\nabla_{\theta} J(\theta)$



Batch Gradient Descend

Формула пересчёта:

$$\theta_t = \theta_{t-1} - \eta \nabla_{\theta} J(\theta_{t-1})$$

- Требуется обработать все объекты для одного шага
- Нет режима online обучения
- Гарантируется сходимость к (локальному) минимуму при правильном выборе шага

SGD / Mini-batch SGD

Большие суммы функций:

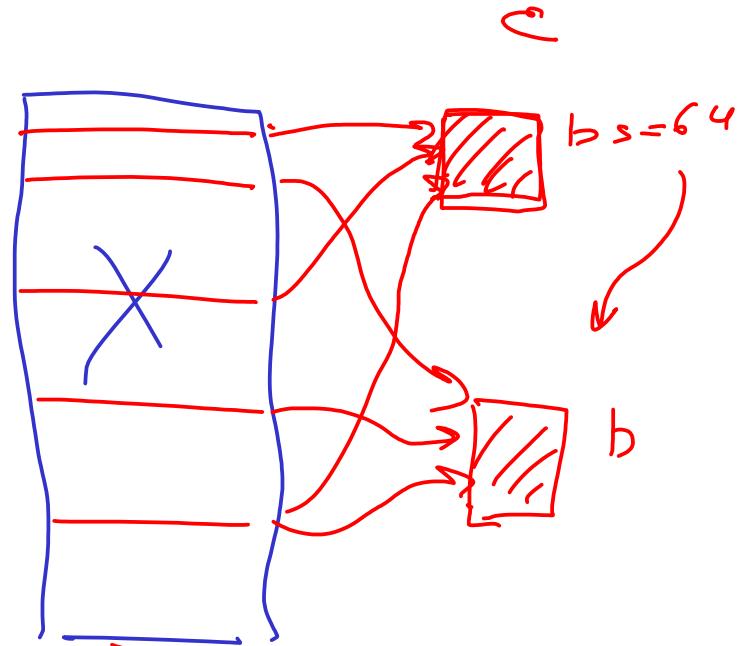
$$J(\theta) = \sum_{i=1}^N J_i(\theta)$$

Формула пересчёта:

$$\theta_t = \theta_{t-1} - \eta \nabla_{\theta} J_i(\theta_{t-1})$$

Mini-batch SGD:

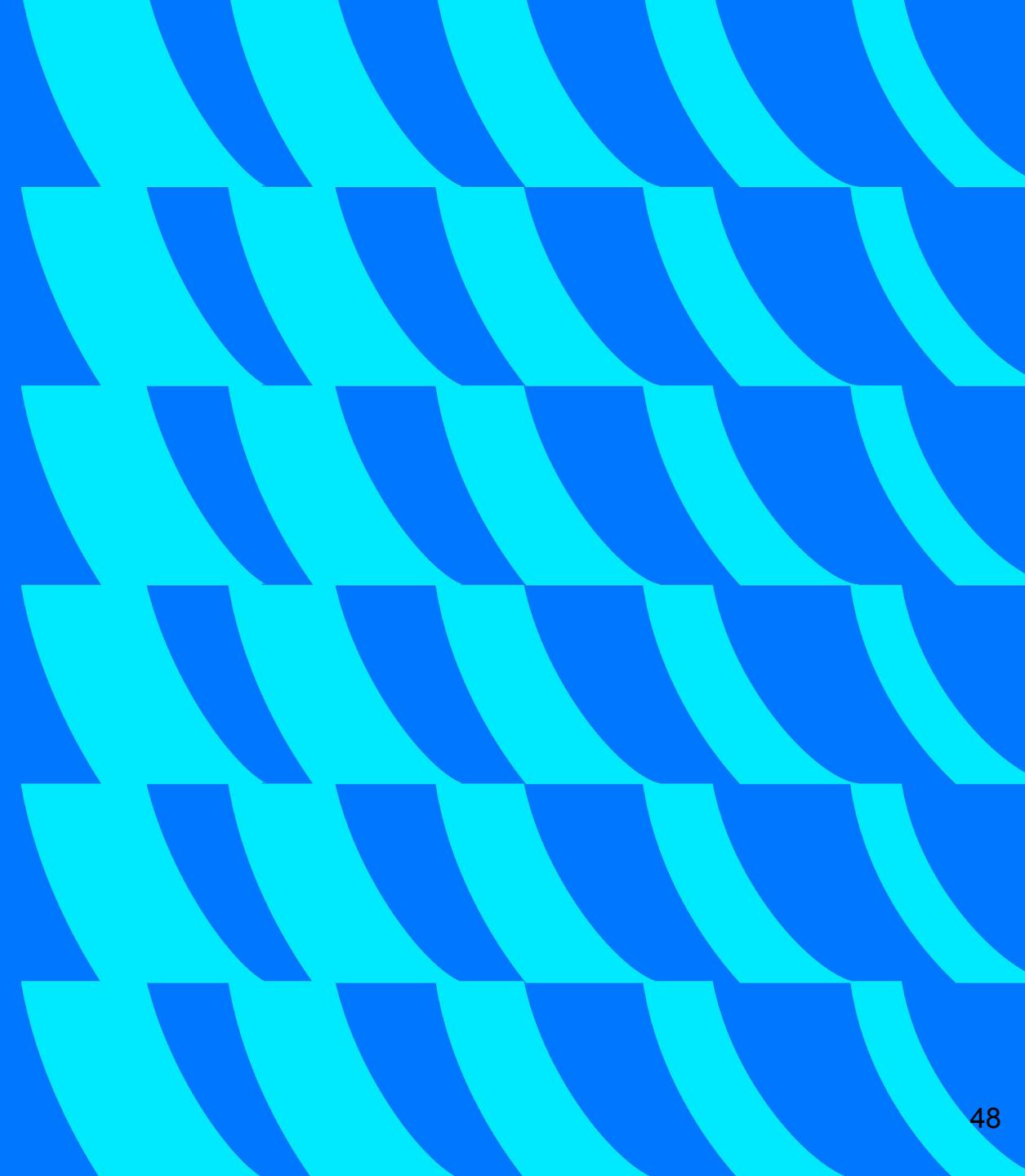
$$\theta_t = \theta_{t-1} - \eta \sum_{i \in \{i_1, i_2, \dots, i_k\}} \nabla_{\theta} J_i(\theta_{t-1})$$



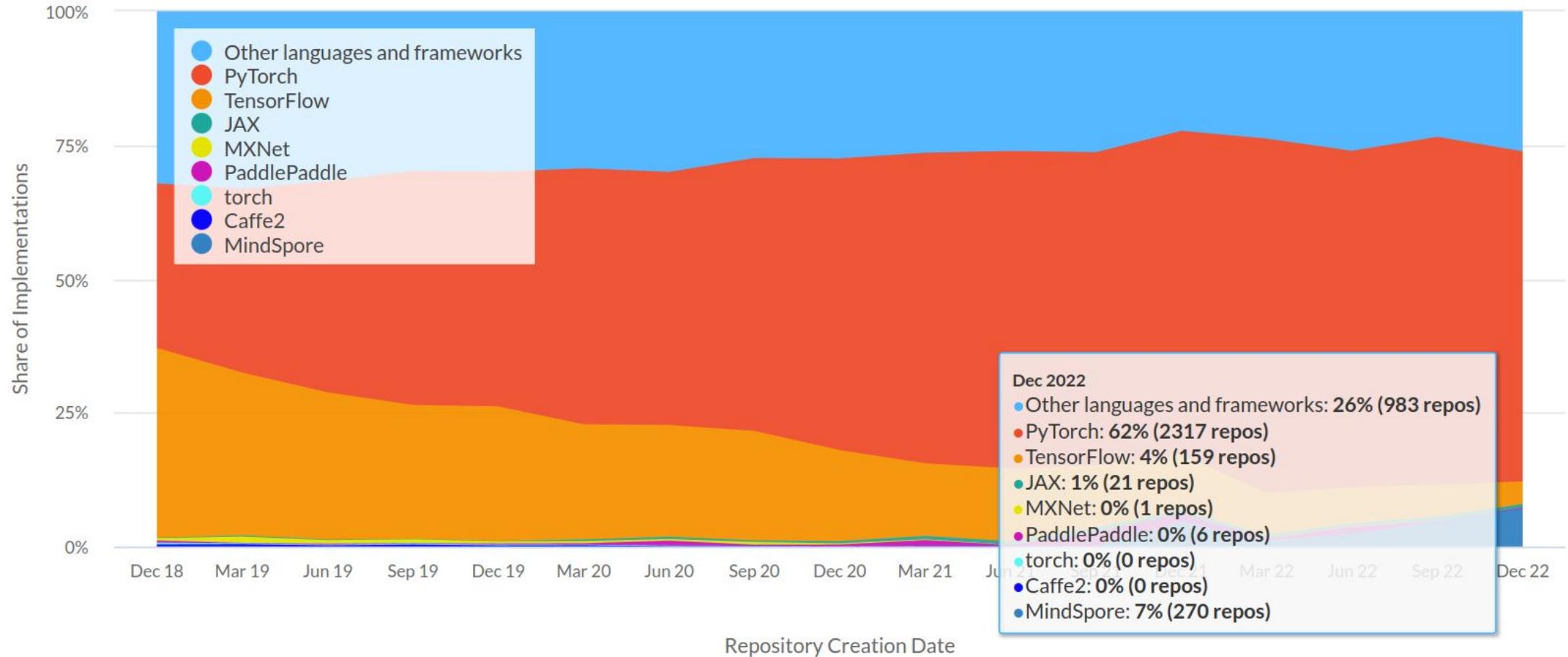
Переобучение: Перемешивание примеров

1. Рекомендуется перемешивать данные перед каждой эпохой;
2. Батчи должны содержать данные как можно большего числа различных классов;
3. Имеет смысл чаще показывать экземпляры, на которых допускается большая ошибка. } Следует быть аккуратным в присутствии выбросов.

PyTorch



PyTorch



<https://pytorch.org/get-started/locally/>

PyTorch

1. Открытый фреймворк для построения и использования динамических графов вычислений и глубокого обучения
2. Есть альтернативы: TensorFlow, JAX, Caffe
3. Изначально разрабатывался Facebook's AI Research Lab (FAIR).
4. Вместе с функционалом Python удобен для экспериментов и разработки

PyTorch

1. Автоматическое дифференцирование
2. Вычисления на базе многомерных матриц (тензоров) - очень похож на питону
3. Поддержка динамических вычислительных графов (создаются при работе)
4. Поддержка вычислений на GPU
5. Есть полезные модули (например, torchvision)

Рабочее окружение

```
import torch # заметьте, что не import pytorch
from platform import python_version
print(python_version()) # 3.9.6 ←
print(torch.version) # 1.10.0 ←
print(torch.cuda.is_available()) # True
```

3.9.6

1.10.0

True

nvidia
AMD

Colab

```
{ from google.colab import drive  
drive.mount("/content/gdrive", force_remount=True)  
data_path = "/content/gdrive/My Drive/Colab Notebooks/name/"  
train_ann_path = data_path + 'train.csv'  
train_df = pd.read_csv(train_ann_path)  
print(train_df.head()) }
```

```
# команды для bash пишутся с !  
!ls /content/gdrive/My\ Drive/Colab\ Notebooks/name/
```

<https://colab.research.google.com>

Тензоры (torch.Tensor)

1. Аналоги многомерных массивов пакета питчу, только могут располагаться на GPU (или поддерживать вычисления на нескольких CPU), могут быть элементами вычислительного графа и поддерживать автоматическое дифференцирование.
2. Это фундаментальная структура данных в Pytorch (с помощью неё будут храниться и обрабатываться объекты: тексты, сигналы изображения и батчи - наборы объектов).
3. Могут в многомерном матричном виде хранить данные определённого типа.

Тензоры (torch.Tensor)

```
x = torch.tensor([[1, 2, 3], [4, 5, 6]])  
print(x, '\n', x.shape, '\n', # размер тензора  
      x.dtype, '\n', # тип  
      x.device, '\n', # где лежит  
      x.type(), '\n', # тип  
      x.dim(), '\n', # размерность  
      x.size(), '\n', # размер; .shape и .size() одно и то же  
      x.numel()) # тип тензора
```

```
tensor([[1, 2, 3],  
       [4, 5, 6]])  
torch.Size([2, 3])  
torch.int64  
cpu  
torch.LongTensor  
2  
torch.Size([2, 3])  
6
```

Тензоры (torch.Tensor)

Создание

```
x = torch.cuda.FloatTensor(2, 3) # те тензоры были на CPU  
print (x)
```

Приведение типов

```
x = torch.IntTensor([1, 2]).float() ✓ 1  
print (x)  
x = torch.IntTensor([1, 2]).to(torch.float64) 2  
print (x)  
x = torch.IntTensor([1, 2]) + 0.0 # 3  
print (x)
```

Тензоры (torch.Tensor)

Во всех функциях ниже создания тензоров есть параметры
dtype - тип элементов тензора и device - где размещать тензор.

```
x = torch.empty(3, 5) # пустая матрица (тензор)
```

```
tensor([-7.0703e+29, 4.5576e-41, 1.0221e-22, 4.5577e-41, 1.0328e-22],  
       [ 4.5577e-41, 1.0462e-22, 4.5577e-41, 1.0060e-22, 4.5577e-41],  
       [ 1.0108e-22, 4.5577e-41, 1.0007e-22, 4.5577e-41, 9.8701e-23]))
```

```
x = torch.ones(3, 5) # матрица из 1
```

```
tensor([[1, 1, 1, 1, 1],  
       [1, 1, 1, 1, 1],  
       [1, 1, 1, 1, 1]])
```

```
x = torch.full((3, 5), 3.14, dtype=torch.float) # матрица из 3.14
```

3.14

Тензоры (torch.Tensor)

Создание

```
# единичная матрица (с единицами на главной диагонали)
x = torch.eye(3, 5)
# случайная матрица с элементами равномерно распределёнными на [0, 1]
torch.manual_seed(123)
x = torch.rand(3, 5)
# случайная матрица с нормально распределёнными элементами
torch.manual_seed(123)
x = torch.randn(3, 5)
# случайная матрица с числами от 2 до 4 (не включая)
torch.manual_seed(123)
x = torch.randint(2, 4, (3, 5))
```

Тензоры (torch.Tensor)

```
x = torch.arange(0, 10, 2) # аналог np.arange
```

```
tensor([0, 2, 4, 6, 8])
```

```
x = torch.linspace(0, 10, 3) # аналог np.linspace
```

```
tensor([ 0., 5., 10.])
```

```
x = torch.logspace(0, 1, 3) # аналог np.logspace
```

```
tensor([ 1.0000, 3.1623, 10.0000])
```

сделать тензоры «по образцу» (использовать такой же тип и размеры)

```
torch.empty_like(x), torch.zeros_like(x), torch.ones_like(x)
```

```
(tensor([ 0., 5., 10.]), tensor([0., 0., 0.]), tensor([1., 1., 1.]))
```

Тензоры (torch.Tensor)

Индексация аналогичная принятой в питоне, в частности в питчу: [start:end:step]

```
x = torch.randint(0, 10, (2, 5))
x[0], x[0, :], x[[0], :], x[:1, :]
```

```
(tensor([6, 5, 3, 9, 4]),
 tensor([6, 5, 3, 9, 4]),
 tensor([[6, 5, 3, 9, 4]]),
 tensor([[6, 5, 3, 9, 4]]))
```

```
x[:, [1]], x[:, 1], x[:, -4]
```

```
(tensor([[5],
         [1]]),
 tensor([5, 1]),
 tensor([5, 1]))
```

Тензоры (torch.Tensor)

Индексация

```
print (x[0, 0]) # это тензор 1x1  
print (x[0, 0].item()) # а это уже отдельный элемент
```

tensor(6)

6

Тензоры (torch.Tensor)

Как скопировать тензор

```
x = torch.tensor([[1, 2], [3, 4]])  
# формально можно, например так  
x2 = torch.empty_like(x).copy_(x)  
x2
```

```
tensor([[1, 2],  
       [3, 4]])
```

```
# но лучше так:  
x2 = x.detach().clone()  
x[0, 1] = 10  
x, x2
```

```
(tensor([[1, 10],  
        [3, 4]]),  
 tensor([[1, 2],  
        [3, 4]]))
```

$x_2 = x$

В отличие от `copy_()` при использовании `clone()` через оригинал проносят градиенты, т.к. копия остаётся в графе вычислений

Тензоры (torch.Tensor)

транспонируем – не происходит копирования, используется та же память

```
xt = x.t()  
x[0, 0] = 30  
print ('t() \n', x, '\n', xt)
```

```
t()  
tensor([[30, 2],  
        [ 3, 4]])  
tensor([[30, 3],  
        [ 2, 4]])
```

Тензоры (torch.Tensor)

```
x = torch.tensor([[1, 2], [3, 4]])
y = torch.tensor([[2, 2], [2, 2]])
z = torch.stack((x, y)) # состыковка тензоров (по умолчанию dim=0)
print(z, '\n', z.shape)
```

```
tensor([[[1, 2],
          [3, 4]],
         [[2, 2],
          [2, 2]]])
torch.Size([2, 2, 2])
```

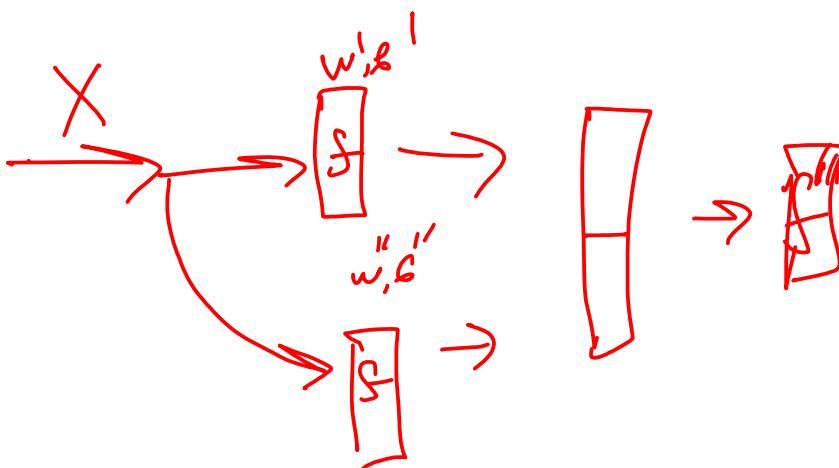
```
x, y = z.unbind(dim=0) # разстыковка тензоров
print(x, '\n', y)
```

```
tensor([[1, 2],
          [3, 4]])
tensor([[2, 2],
          [2, 2]])
```

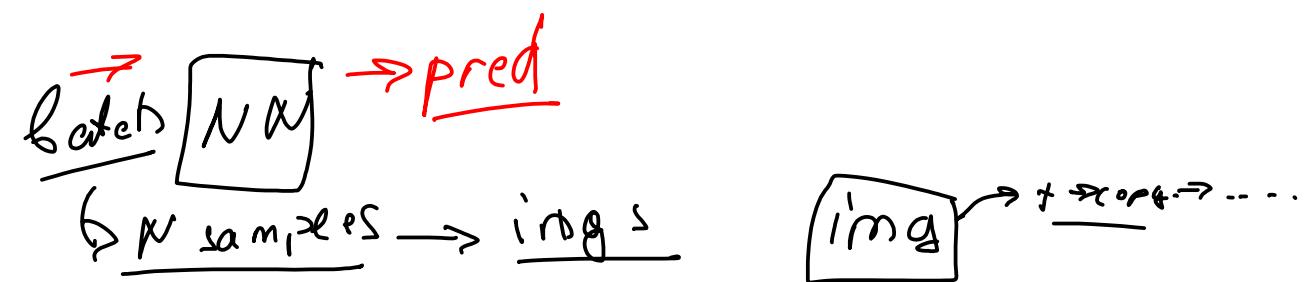
Тензоры (torch.Tensor)

```
# конкатенация по 0 и 1 размерностям
x = torch.tensor([[1, 2], [3, 4]])
y = torch.tensor([[2, 2], [2, 2]])
torch.cat([x, y], axis=1),
torch.cat([x, y], axis=0)
```

```
(tensor([[1, 2, 2, 2],
        [3, 4, 2, 2]]),
 tensor([[1, 2],
        [3, 4],
        [2, 2],
        [2, 2]]))
```



Тензоры (torch.Tensor)



```
# создание фиктивной размерности - в какую позицию вставлять фиктивную  
x.unsqueeze(dim=0).shape, x.unsqueeze(dim=1).shape, x.unsqueeze(dim=2).shape
```

~~img~~
(`torch.Size([1, 2, 2]), torch.Size([2, 1, 2]), torch.Size([2, 2, 1])`)

```
# удаляем единичные размеры  
torch.empty(3, 1, 2, 1).squeeze().shape
```

`torch.Size([3, 2])`

Тензоры

1. Можно менять «представление» тензора с помощью `view`, фактически это изменение размеров, но реально данные не перемещаются, pytorch просто запоминает, что тензор, заданный элементами, лежащими в определённой области памяти, имеет другой размер.
 - a. `View` работает только на `contiguous tensors` (которые «правильно» последовательно лежат в памяти)
 - b. При использовании `view` может выдаваться сообщение об ошибке – если нельзя
 - c. использовать эту область памяти (можно тогда сделать предварительно `.contiguous()`)
2. `Reshape` работает всегда (старается выдать `view`, если не получается делает копию данных). При `reshape()` тензор может копироваться!

Тензоры

```
x = torch.arange(4*10*2).view(4, 10, 2)
y = x.permute(2, 0, 1)
# View
print('смежность', x.is contiguous())
print('вытягиваем', x.view(-1))
```



смежность True

вытягиваем tensor([0, 1, ...

```
# Reshape
print('смежность', y.is contiguous())
print('вытягиваем', y.view(-1))
```

смежность False

ошибка view size is not compatible with input tensor's size and stride

```
print('решейпим', y.reshape(-1))
print('делаем смежным и решейпим', y.contiguous().view(-1))
```

решейпим tensor([0, 2, 4, ...

делаем смежным и решейпим tensor([0, 2, 4, ...

Тензоры

```
x = torch.arange(8)  
x.view(4, 2), x.view(2, -1)
```

```
(tensor([[0, 1],  
        [2, 3],  
        [4, 5],  
        [6, 7]]),  
 tensor([[0, 1, 2, 3],  
        [4, 5, 6, 7]]))
```

- Здесь сам тензор не поменялся, т.к. не было присваивания $x = x.view()$
- В Pytorch-е тензоры хранятся в формате [channel, height, width], в других системах чаще [height, width, channel].

Тензоры

```
# как хранятся данные, где следующий элемент по каждой из размерностей  
print (x, x.storage(), x.stride(), x.t().stride())
```

```
tensor([[0, 1],  
        [2, 3],  
        [4, 5],  
        [6, 7]])
```

```
0  
1  
2  
3  
4  
5  
6  
7
```

```
[torch.LongStorage of size 8] (2, 1) (1, 2)
```

Тензоры

```
z = torch.rand(1, 2, 3, 4)
z = z.permute(0, 3, 1, 2) # NxHxWxC -> NxCxHxW
z.shape
```

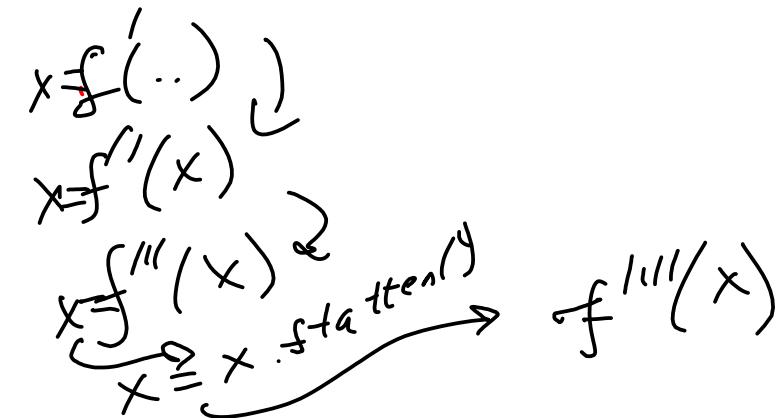
`torch.Size([1, 4, 2, 3])`

```
# другие способы транспонирования:
x.transpose(0, 1), x.t(), x.t_()
```

```
(tensor([[0, 2, 4, 6],
         [1, 3, 5, 7]]),
 tensor([[0, 2, 4, 6],
         [1, 3, 5, 7]]),
 tensor([[0, 2, 4, 6],
         [1, 3, 5, 7]]))
```

```
# векторизация
x.flatten() # ещё вариант .view(-1)
```

`tensor([0, 2, 4, 6, 1, 3, 5, 7])`



Тензоры. Операции

1. Операции аналогичны операциям в питчу, большинство операций выполняются поэлементно.
2. Поддерживаются операции линейной алгебры, многие из которых взяты из библиотек Basic Linear Algebra Subprograms (BLAS) и Linear Algebra Package (LAPACK).
3. Полный список операций линейной алгебры:

<https://pytorch.org/docs/stable/torch.html#blas-and-lapack-operations>

Тензоры. Операции

В `inplace`-операциях используется черта, в этом случае операция выполняется на данном тензоре:

```
x = torch.tensor([[1, 2], [3, 4]])  
# заполнение  
x.fill_(3) # черта - признак  
выполнения на данном тензоре  
# обнуление  
x.zero_()
```

`inplace`-операции специально «запрятаны», так как при их использовании могут быть проблемы при распространении градиента.

Тензоры. Операции

```
x = torch.tensor([[1, 2], [3, 4]])  
y = torch.tensor([[2, 2], [2, 2]])  
v = torch.tensor([1, 2])
```

```
# сложение  
print(x + y,  
x.add(y))  
# черта означает inplace-операцию - меняется первый тензор:  
x.add_(y)
```

```
# поэлементное умножение  
x * y, x.mul(y), torch.mul(x, y)
```

Тензоры. Операции

Матричное умножение можно сделать по-разному:

1. `torch.matmul` – операция определена над тензорами, можно указывать размерность для умножения (см)
2. `torch.mm` – обычное матричное умножение, но без приведения размеров (broadcasting)
3. `torch.bmm` – матричное умножение с поддержкой батчей: $(b \times n \times m) (b \times m \times p) = b \times n \times p$

```
# матричное умножение
x @ y, x.mm(y), x.matmul(y), torch.matmul(x, y)
```

```
(tensor([[ 6,  6],
          [14, 14]]),
 tensor([[ 6,  6],
          [14, 14]]),
 tensor([[ 6,  6],
          [14, 14]]),
 tensor([[ 6,  6],
          [14, 14]]))
```

Тензоры. Операции

И многие другие операции:

1. Специальные операции типа $y + x^*x \dots$
2. Определитель, с.з., ...
3. Решение уравнений, SVD, ...

Тензоры. Статистики

```
torch.max(x), x.max(), x.max().item(), x.max(axis=0), x.max(axis=1)
```

```
(tensor(4),
tensor(4),
4,
torch.return_types.max(
values=tensor([3, 4]),
indices=tensor([1, 1])),
torch.return_types.max(
values=tensor([2, 4]),
indices=tensor([1, 1]))
)
```

Тензоры. Статистики

```
x = torch.tensor([2, 1, 2, 3, 0, 4, 3])
x.topk(k=2), x.kthvalue(k=2)
```

```
(torch.return_types.topk(
    values=tensor([4, 3]),
    indices=tensor([5, 3])),
torch.return_types.kthvalue(
    values=tensor(1),
    indices=tensor(1)))
```

```
x = torch.tensor([1., 2., 3.])
# сразу вычислить СКО / дисперсию и среднее
torch.std_mean(x), torch.var_mean(x)
```

```
((tensor(1.), tensor(2.)), (tensor(1.), tensor(2.)))
```

Тензоры. Операции

1. когда 2 размерности совпадают
2. когда одна размерность равна 1

```
x = torch.tensor([[1, 2], [3, 4]])  
y = torch.tensor([10, 20])  
x + y
```

```
tensor([[11, 22],  
       [13, 24]])
```

Тензоры. Операции

```
x = torch.tensor([[1, 2]])
y = torch.tensor([[0], [3]])
print(x, x.shape)
print(y, y.shape)
z = x + y
print(z, z.shape)
```

```
tensor([[1, 2]]) torch.Size([1, 2])
tensor([0,
       3]) torch.Size([2, 1])
tensor([[1, 2],
       [4, 5]]) torch.Size([2, 2])
```

Тензоры. Операции

При переводе в питону необходимо, чтобы тензор находился в CPU, а не на GPU.

```
x = torch.tensor([[1, 2], [3, 4]])  
x.numpy()
```

```
array([[1, 2],  
       [3, 4]], dtype=int64)
```

```
# из numpy.array в pytorch-тензор  
torch.from_numpy(np.ones((2, 2)))
```

```
tensor([[1., 1.],  
       [1., 1.]], dtype=torch.float64)
```

Тензоры. Операции

```
# сохранение и загрузка тензоров
torch.save(x, 'x-file')
x2 = torch.load("x-file")
x2
```

```
mydict = { 'x' : x, 'y' : y }           x , y
torch.save(mydict, 'mydict')
mydict2 = torch.load('mydict')
mydict2
```

```
{'x': tensor([1., 2., 3.]),
'y': tensor([[ 3., 0.],
            [ 3., 12.]]), requires_grad=True)}
```

Тензоры. GPU

1. Тензоры и вычислительные графы (нейросети) могут находиться как на CPU, так и на GPU (ещё на TPU).
2. Переменные и модели на разных устройствах не видят друг друга! Поэтому их надо перенести на один вычислитель.
3. На GPU вычисления производятся существенно быстрее из-за параллелизма.

Такие записи эквивалентны:

```
device="cuda"  
device=torch.device("cuda")  
device="cuda:0" ←  
device=torch.device("cuda:0")
```

```
x.cuda()  
x.cpu()  
x.is_cuda  
z.to("cpu", torch.double)
```

Тензоры. GPU

```
# самая популярная конструкция, определяющая
device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")
m.to(device) # перенос на доступное устройство
```

```
x = torch.randn(5000, 5000)
# CPU
_ = torch.matmul(x, x)
```

CPU time: 0.77800s

```
# 2GPU
x = x.to("cuda:0")
```

CPU2GPU time: 0.09263s

```
# GPU
x = x + 0.0 # просто первая операция
_ = torch.matmul(x, x)
```

GPU time: 0.00803s

Тензоры

Инициализация генератора псевдослучайных чисел

```
def set_seed(seed):  
    np.random.seed(seed)  
    torch.manual_seed(seed)  
    if torch.cuda.is_available(): # для GPU отдельный seed  
        torch.cuda.manual_seed(seed)  
        torch.cuda.manual_seed_all(seed)  
  
set seed(42)  
# есть стохастические операции на GPU  
# сделаем их детерминированными для воспроизводимости  
torch.backends.cudnn.deterministic = True  
torch.backends.cudnn.benchmark = False
```

Авто дифференцирование (AutoGrad)

Рассмотрим пример автоматического дифференцирования

Зададим функцию

$$y = \sin(x) \cdot (\underbrace{\sin^2(x)}_{-} + \underbrace{\cos^2(x)}_{\sim})$$

Pytorch автоматически вычислит её производную backward – функция «обратного прохода», именно при её вызове автоматически считаются производные по всем переменным, у которых `requires_grad=True`

Авто дифференцирование (AutoGrad)

```
from torch.autograd import Variable
x = torch.linspace(-2, 2, 101, dtype=torch.float32, requires_grad=True)
# x = Variable(x, requires_grad=True) # можно ли проще? - да!
y = torch.sin(x) * (torch.sin(x) ** 2 + torch.cos(x) ** 2)
# здесь это прямой проход
# дальше часто будем определять класс с методом y.forward()
y.sum().backward() # превращаем в число
# (только от таких функций берётся градиент)
g = x.grad # взятие производных в каждой точке
```

```
x = torch.linspace(-2, 2, 3, dtype=torch.float32)
# над целочисленными тензорами нельзя взять производную
x.requires_grad_() # как указать, что хотим вычислять производную
x
```

```
tensor([-2., 0., 2.], requires_grad=True)
```

Авто дифференцирование (AutoGrad)

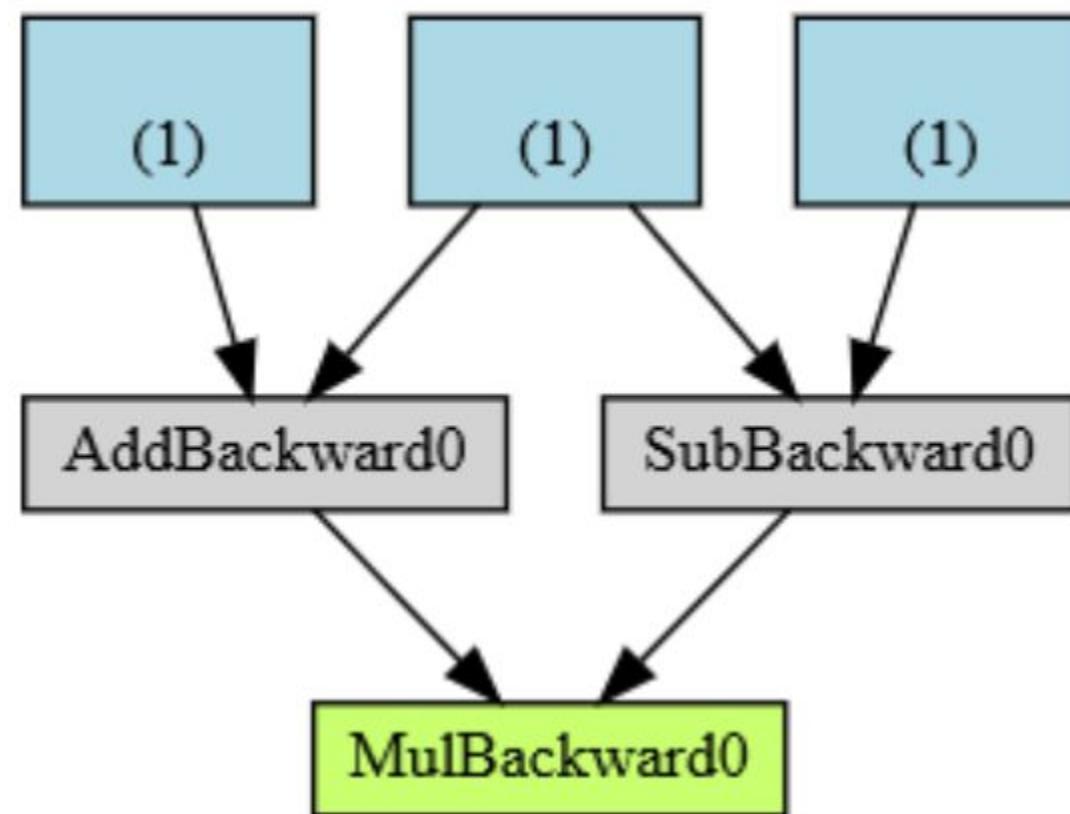
```
import torch
from torch.autograd import Variable

# x = Variable(torch.Tensor([1]), requires_grad=True)
# y = Variable(torch.Tensor([2]), requires_grad=True)
# z = Variable(torch.Tensor([3]), requires_grad=True)
x = torch.tensor([1.], requires_grad=True)
y = torch.tensor([2.], requires_grad=True)
z = torch.tensor([3.], requires_grad=True)
f = (x + y) * (y - z)
f.backward()
x.grad, y.grad, z.grad
```

```
from torchviz import make_dot
make_dot(f)
```

Авто дифференцирование (AutoGrad)

Граф вычислений



Авто дифференцирование (AutoGrad)

Если устраним вычисления градиентов - считается быстрее

```
f = (x + y) * (y - z)
z = f.detach()
print(f.requires_grad, z.requires_grad)

with torch.no_grad(): # когда просто нужен прямой проход -
    # и не надо считать градиенты
f = (x + y) * (y - z)
print(f.requires_grad)
```

True False

False

Авто дифференцирование (AutoGrad)

$$z = w_1 x_1 + w_2 x_2 + w_3 x_3$$

```
x = torch.Tensor([1., 2., 3])
w = torch.tensor([1., 1., 1], requires_grad=True)
z = w @ x
z.backward()
print(x.grad, # поэтому не указывали возможность взятия градиента
      w.grad, # должен выводиться x
      sep='\n')
```

$$\frac{\partial z}{\partial w_1} = x_1, \quad \frac{\partial z}{\partial w_2} = x_2 \quad \text{и} \quad \frac{\partial z}{\partial w_3} = x_3$$

None

tensor([1., 2., 3.])

```
z = w @ x
z.backward()
print(x.grad,
      w.grad, # идёт накопление!!!
      sep='\n')
```

$$z = \sum \frac{\partial z}{\partial w}$$

None

tensor([2., 4., 6.])

Авто дифференцирование (AutoGrad)

```
with torch.no_grad(): # нет накопления
    z = w @ x
    # z.backward()
    print(x.grad, w.grad, sep='\n')
w.grad.data.zero_() # а так - совсем обнулить
z = w @ x
z.backward()
print(x.grad, w.grad, sep='\n')
```

None

tensor([2., 4., 6.])

None

tensor([1., 2., 3.])

```
# w.numpy() - будет ошибка
w.detach().numpy() # создаётся копия, которую можно впр-
                                    # у неё requires_grad=False
```

array([1., 1., 1.], dtype=float32)

Авто дифференцирование (AutoGrad)

```
# Иллюстрация взятия градиента - с detach
x = torch.tensor([2.], requires_grad=True)
y = x * x
y.detach_() # добавили
z = x * y
# а тут не сработает
z.backward()
print(x.grad) # (2*2*x)' = 4
```

tensor([4.])

Организация подачи данных в модель

1. TensorDataset – для представления датасета. Часто приходится определять свой датасет, наследуя этот класс.
2. DataLoader – подаёт батчами данные, позволяет итерироваться по датасету, автоматически формируя батчи (и делая некоторые сопутствующие действия).

В DataLoader может передаваться сэмплер.

```
from torch.utils.data import TensorDataset  
import numpy as np  
from torch.utils.data import DataLoader
```

Организация подачи данных в модель

```
x = torch.from_numpy(np.vstack([np.arange(10, dtype='float32'),  
                                np.ones(10, dtype='float32')]).T)  
y = torch.from_numpy(np.arange(10, dtype='float32')[:, np.newaxis] ** 2)  
train_ds = TensorDataset(x, y)  
train_dl = DataLoader(train_ds, batch_size=4, shuffle=True)  
for xb, yb in train_dl:  
    print(xb)  
    print(yb)
```

tensor([[0., 1.],
 [8., 1.],
 [9., 1.],
 [7., 1.]])

tensor([[0.],
 [64.],
 [81.],
 [49.]])

tensor([[1., 1.],
 [5., 1.]])

tensor([[1.],
 [25.]])

tensor([[4., 1.],
 [3., 1.],
 [6., 1.],
 [2., 1.]])

tensor([[16.],
 [9.],
 [36.],
 [4.]])

Организация подачи данных в модель

```
train_loader = torch.utils.data.DataLoader(  
    datasets.MNIST('../data', train=True, download=True,  
    transform=transforms.Compose([  
        1 transforms.ColorJitter(brightness=(0.6, 1),  
                                 contrast=(0.8, 1)),  
        2 transforms.RandomRotation((-15, 15)),  
        # перевод в тензор + нормировка на отрезок  
        3 transforms.ToTensor(),  
        4 transforms.Normalize((0.1307,), (0.3081,))  
    ])),  
    batch_size=64, shuffle=True, num_workers=8, pin_memory=True)
```

- С `num_workers` лучше поэкспериментировать! Это число подпроцессов для загрузки данных.
- Чаще `pin_memory=True` если используем GPU. Предварительно копирует данные на GPU

Организация подачи данных в модель

```
trainset = torchvision.datasets.CIFAR10(root='./data', train=True,  
                                         download=True, transform=transform)  
testset = torchvision.datasets.CIFAR10(root='./data', train=False,  
                                         download=True, transform=transform)
```

- Трансформации можно и нужно делать разные для обучения и теста

Организация подачи данных в модель

```
class Noise():
    """
    Adds gaussian noise to a tensor.
    """
    def __init__(self, mean, stddev):
        self.mean = mean
        self.stddev = stddev

    def __call__(self, tensor):
        noise = torch.zeros_like(tensor).normal_(self.mean, self.stddev)
        return tensor.add_(noise)

    def __repr__(self):
        repr = f'{self.__class__.__name__}(mean={self.mean}, stddev={self.stddev})'
        return repr
```

- Своя трансформация

Организация подачи данных в модель

```
class CustomTextDataset(Dataset):
    """
    Simple Dataset initializes with X and y vectors
    """
    def __init__(self, X, y=None):
        self.data = list(zip(X, y))
        # Sort by length of first element in tuple
        self.data = sorted(self.data, key=lambda x: len(x[0]))
```

1) def len (self):
 # raise NotImplementedError
 return len(self.data)

2) def __getitem__ (self, idx):
 # raise NotImplementedError
 return self.data[idx]

- Свой датасет

Модуль nn

Простейший линейный слой

```
from torch import nn

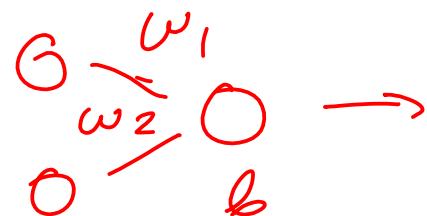
model = nn.Linear(in_features=2,
                  out_features=1,
                  bias=True)
print(model.weight)
print(model.bias)
```

Parameter containing:

tensor([[-0.1357, -0.0438]], requires_grad=True) ←

Parameter containing:

tensor([-0.0287], requires_grad=True) ←



Модуль nn

```
# через nn.Sequential  
net = nn.Sequential(nn.Linear(10, 5),  
                     nn.ReLU(),  
                     nn.Linear(5, 2))  
                     nn.ReLU() —>  
nn.Linear(5, 2)  
an.ReLU() —>  
Sigmoid(5, 2)
```

```
# через nn.Sequential, но с удобными именами  
from collections import OrderedDict
```

```
net1 = nn.Sequential(OrderedDict([('hidden linear', nn.Linear(10, 5)),  
                                 ('hidden activation', nn.ReLU()),  
                                 ('output', nn.Linear(5, 2))]))
```

```
X = torch.rand(3, 10)
```

```
net(X)
```

```
tensor([[ 0.2508, -0.0461],  
        [ 0.2470, -0.1381],  
        [ 0.2298, -0.2766]], grad_fn=<AddmmBackward0>)
```

Модуль nn. MLP

Наследуя от nn.Module и прописывая `__init__` и `forward`

```
import torch.nn.functional as F

class MLP(nn.Module):
    def __init__(self):
        # инициализация параметров
        # обращение к инициализации родителя
        super().__init__() # super(MLP, self). init_()
        self.hidden = nn.Linear(10, 5) # Hidden layer
        self.out = nn.Linear(5, 2) # Output layer

    def forward(self, X):
        # как обрабатываются данные и получается ответ
        return self.out(F.relu(self.hidden(X)))
```

$$f'(f^2(f^3(x)))$$

$$g'(g^2(g^3 x))$$

Модуль nn

Присоединение модулей с помощью .add_module

```
class NNN(torch.nn.Module):
    def __init__(self):
        super(NNN, self).__init__()
        self.layers = torch.nn.Sequential()
        self.layers.add_module('lin1', torch.nn.Linear(10, 5))
        self.layers.add_module('relu1', torch.nn.ReLU())
        self.layers.add_module('lin2', torch.nn.Linear(5, 2))

    def forward(self, input):
        return self.layers(input)
```

Модуль nn

```
print(net.state_dict()) # все параметры сети

OrderedDict([('hidden_linear.weight',
 tensor([[ 0.3100,  0.2531, -0.1647,  0.3126,  0.1476,  0.0046, -0.0384, -0.1415, -0.0294,  0.2909],
 [ 0.1773, -0.1971,  0.0959,  0.2161, -0.3102,  0.1470,  0.2708, -0.0020,  0.0557,  0.2001],
 [ 0.0238, -0.2250, -0.1456, -0.2171,  0.0509,  0.2531, -0.2238, -0.0303,  0.2203, -0.0681],
 [-0.2177,  0.2239,  0.1431, -0.0973,  0.3137, -0.2519, -0.0794, -0.2917,  0.0505,  0.1818],
 [ 0.2810, -0.0352,  0.1037,  0.3152,  0.1919,  0.1356,  0.0720, -0.0077, -0.0081, -0.3075]]),
('hidden_linear.bias',
 tensor([ 0.0292,  0.2533,  0.2812, -0.0061, -0.1667])),
('output.weight',
 tensor([[-0.1961,  0.0860,  0.3077, -0.3915,  0.0169],
 [ 0.3430, -0.0144,  0.1403, -0.1678, -0.3166]])),
('output.bias', tensor([-0.4303,  0.2812]))])
```

Модуль nn

Инициализация весов сети

```
def init_normal(m):
    if type(m) == nn.Linear:
        nn.init.normal_(m.weight, mean=0, std=0.01)
        nn.init.zeros_(m.bias)

net.apply(init_normal)
net[0].weight.data[0], net[0].bias.data[0]
```

Можно применять и по отдельным модулям

Обучение сети

Присоединение модулей с помощью .add_module

```
def train_epoch(model, train_loader, criterion, optimizer):  
    model.train()  
    running_loss = 0.0  
    for batch_idx, (data, target) in enumerate(train_loader):  
        optimizer.zero_grad(set_to_none=True) # ~ model.zero_grad()  
        # = 0 чтобы не накапливались  
        data = data.to(device)  
        target = target.to(device) # перенос на device  
        outputs = model(data) # получили выход сети  
        loss = criterion(outputs, target) # посчитали для этого выхода лосс  
        running_loss += loss.item()  
        loss.backward() # вычислили градиенты loss по параметрам сети (w)  
        optimizer.step() # сдадем шаг по антиградиенту - обновляем веса сети.  
        running_loss /= len(train_loader)  
    return running_loss
```

outputs $\hat{y}(x)$
 y_{pred}
 y_{true}
 $L = \frac{1}{N} \sum (y_{true} - y_{pred})^2$
 $\frac{\partial L}{\partial w}$

device = CPU
gpu cuda: 0

$w_t' = w_{t-1}' - \eta \frac{\partial L}{\partial w'}$

Обучение сети

```
# [S] CrossEntropyLoss = Softmax + CrossEntropy
# тот же эффект - logSoftmax + NLLLoss
loss = nn.CrossEntropyLoss()
a = torch.tensor([[1.0, 2.0, 3.0]])
y = torch.tensor([1])
print(a, y, loss(a, y))
```

```
tensor([1., 2., 3.]) tensor([1]) tensor(1.4076)
```

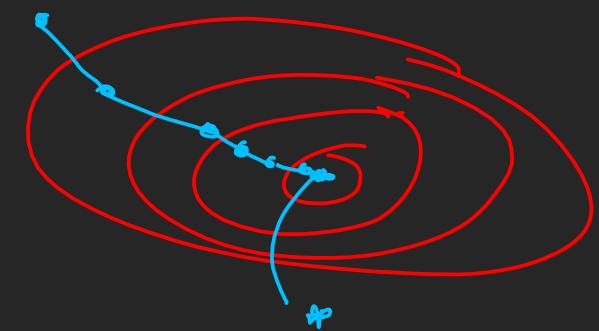
Обучение сети

```
# свой темп для каждого слоя

from torch import optim
optim.SGD([
    {'params': model.features.parameters()},
    {'params': model.classifier.parameters(), 'lr': 1e-3}
],
lr=1e-2, momentum=0.9)
```

Обучение сети

```
from torch.optim.lr_scheduler import StepLR, ReduceLROnPlateau, CosineAnnealingLR
# сообщаем параметры оптимизатору!
optimizer = optim.SGD(net.parameters(), lr)
# сообщаем оптимизатор "шедулеру"
scheduler_1 = ReduceLROnPlateau(optimizer, factor=0.1, patience=1, threshold=0.1)
# умножаем на gamma через каждые step_size шагов
scheduler_2 = StepLR(optimizer, step_size=1, gamma=0.1)
# ....
# шаг на эпоху должен быть один, сделаем его после валидации
if not is train:
    scheduler_1.step(running_loss / (i + 1))
    scheduler_2.step()
```



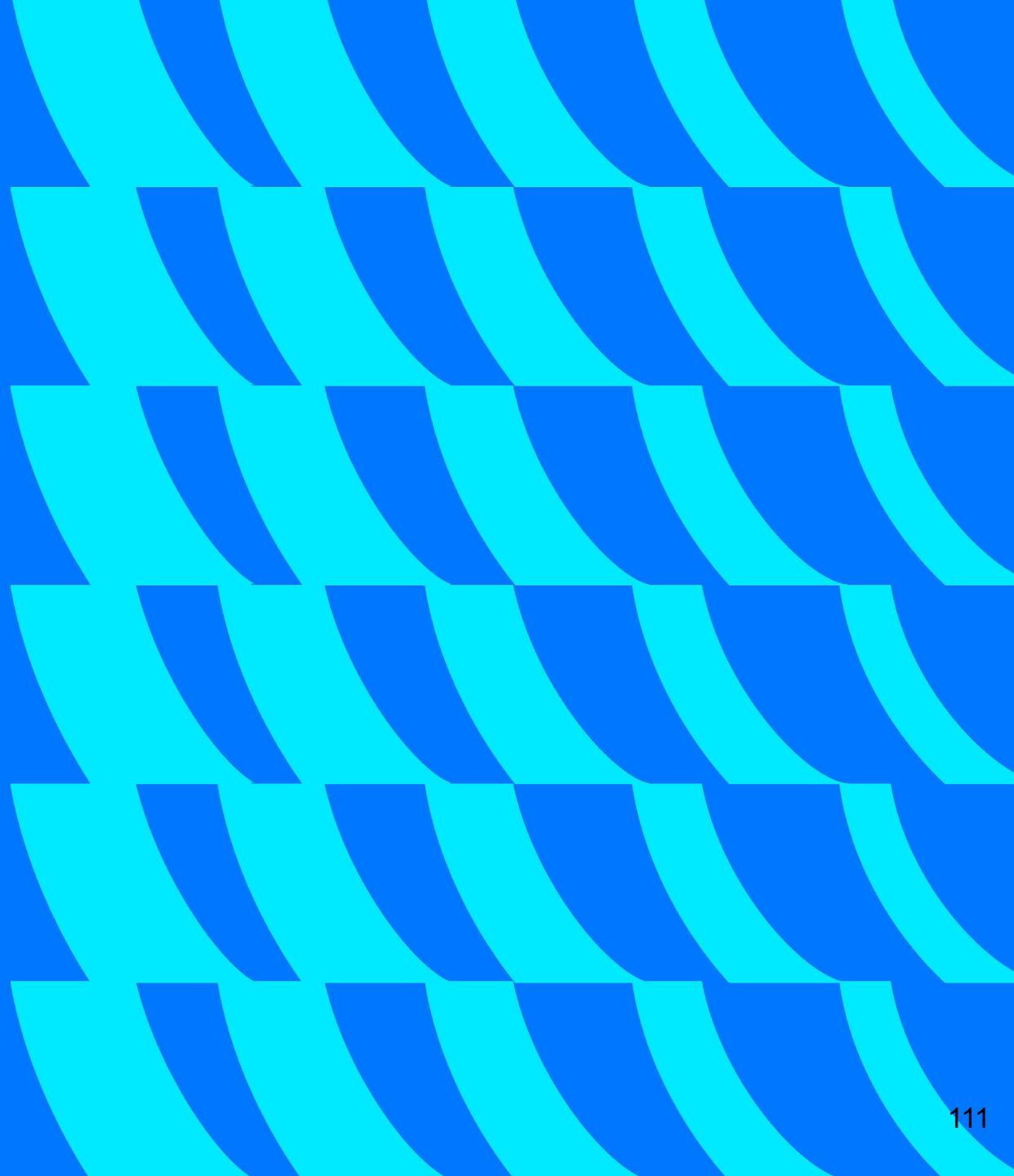
Сохранение / загрузка сети

```
torch.save(model, "/tmp/model.pth")
model = torch.load("/tmp/model.pth")
torch.save(model.state_dict(), "/tmp/model.pth") # только параметры
model = MLP()
model.load_state_dict(torch.load("/tmp/model.pth")) # только параметры
model.to(device) # вот сейчас переносим
model.eval()
```

```
state = {
    'epoch': epoch + 1,
    'state_dict': net.state_dict(),
    'optimizer': optimizer.state_dict()
}
torch.save(state, './my_checkpoint.pth')
```

Домашнее задание

• • • • •



Задание

1. Реализовать Tanh

2. Аугментация

Небольшие вращения (-15, 15)

Случайные сдвиги

Шум

```
network = NeuralNetwork([
    Linear(784, 100), Sigmoid(), # 28 * 28
    Linear(100, 100), Sigmoid(),
    Linear(100, 10)
])

loss = NLLLoss()

def train(network, epochs, learning_rate, plot=True,
          verbose=True, loss=None):
    loss = loss or NLLLoss()
    train_loss_epochs = []
    test_loss_epochs = []
    train_accuracy_epochs = []
    test_accuracy_epochs = []

    try:
        for epoch in range(epochs):
            losses = []
            accuracies = []
            for X, y in train_loader:
                X = X.view(X.shape[0], -1).numpy()
                ...

```



Спасибо
за внимание!