

Лекция 2: Обучение искусственных нейронных сетей в PyTorch

Автор: Сергей Вячеславович Макрушин e-mail: SVMakrushin@fa.ru (<mailto:SVMakrushin@fa.ru>)

Финансовый университет, 2021 г.

При подготовке лекции использованы материалы:

- ...
- v0.5 старое название: TCN20_INNp2_2_v5
- v0.6 18.02.2021

In [1]:

```
# загружаем стиль для оформления презентации
from IPython.display import HTML
from urllib.request import urlopen
html = urlopen("file:./lec_v2.css")
HTML(html.read().decode('utf-8'))
```

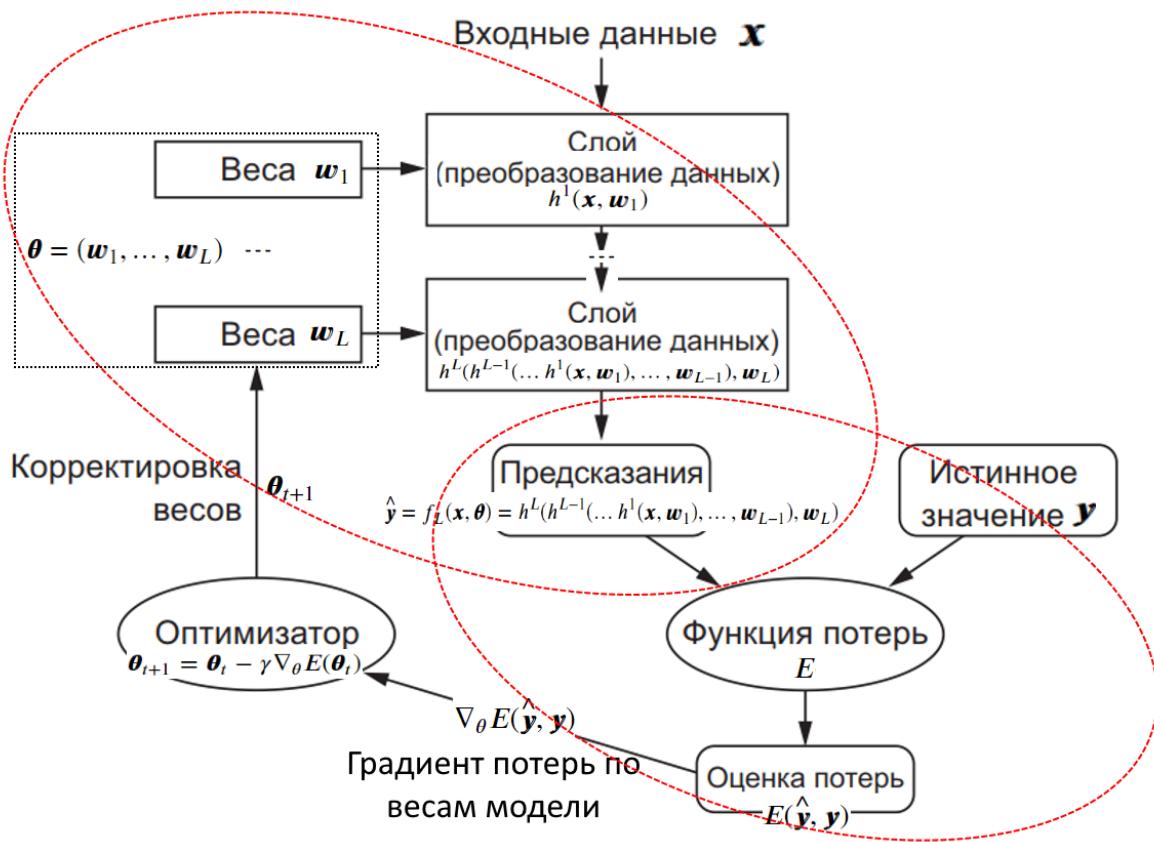
Out[1]:

```
## Современные методы обучения нейронной сети и
обратное распространение ошибки <a class="anchor" id="современные-
методы"></a>
* [к оглавлению] (#разделы)
```

Применение тензоров: прямое распространение сигналов и оценка ошибки

Постановка задачи

- У нас есть набор данных D , состоящий из пар (\mathbf{x}, \mathbf{y}) , где \mathbf{x} - признаки, а \mathbf{y} - правильный ответ.
- Модель сети f_L , имеющей L слоев с весами $\boldsymbol{\theta}$ (совокупность весов нейронов из всех слоев) на этих данных делает некоторые предсказания $\hat{\mathbf{y}} = f_L(\mathbf{x}, \boldsymbol{\theta})$
- Задана функция ошибки E , которую можно подсчитать на каждом примере: $E(f_L(\mathbf{x}, \boldsymbol{\theta}), \mathbf{y})$ (например, это может быть квадрат или модуль отклонения $\hat{\mathbf{y}}$ от \mathbf{y} в случае регрессии или перекрестная энтропия в случае классификации)
- Тогда суммарная ошибка на наборе данных D будет функцией от параметров модели: $E(\boldsymbol{\theta})$ и определяется как $E(\boldsymbol{\theta}) = \sum_{(\mathbf{x}, \mathbf{y}) \in D} E(f_L(\mathbf{x}, \boldsymbol{\theta}), \mathbf{y})$

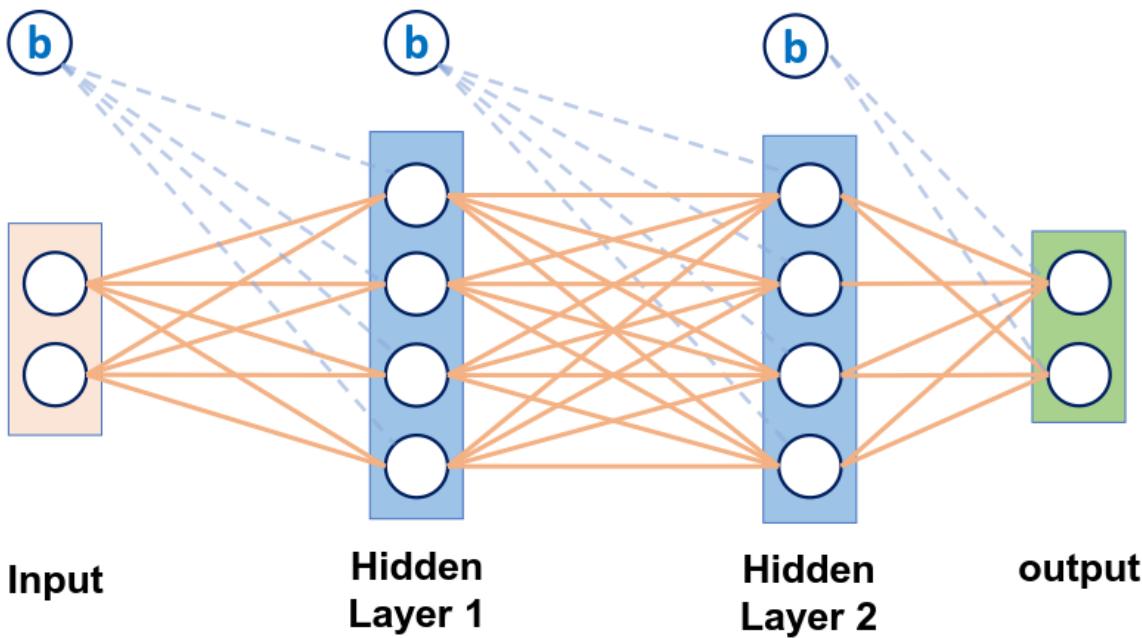


Прямой проход и оценка ошибки

Прямое распространение сигналов

- Модель нейронной сети это иерархия (она может быть простой и очень сложной) связанных (последовательно применяемых) функций слоев:
 - т.е. модель сети f_L может быть представлена как суперпозиция из L слоев h^i , $i \in \{1, \dots, L\}$, каждый из которых параметризуется своими весами \mathbf{w}_i :

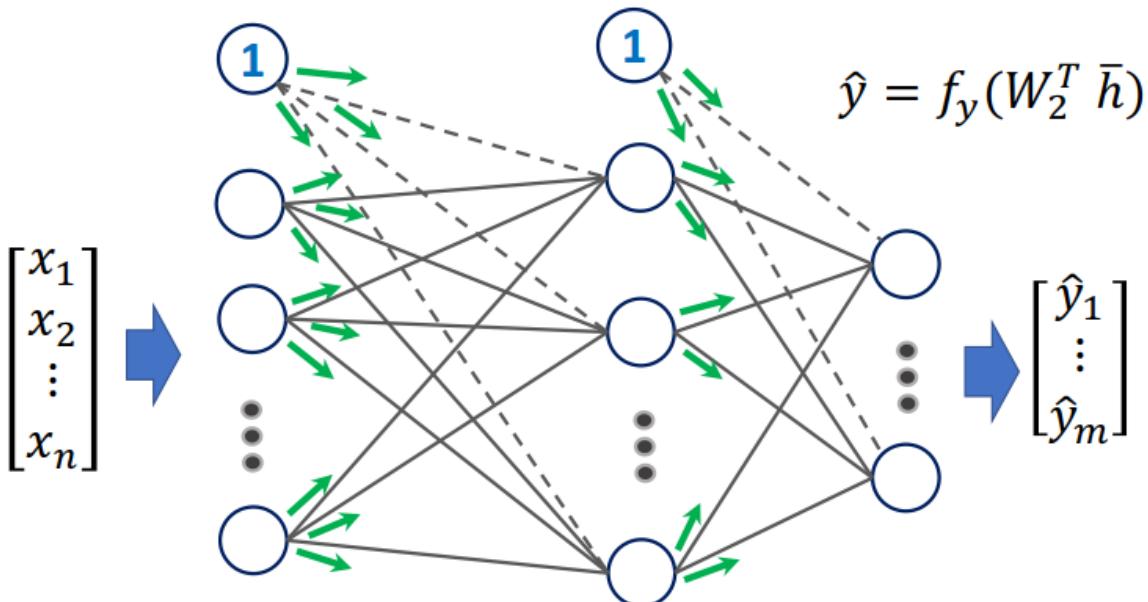
$$f_L(\mathbf{x}, \theta) = f_L(\mathbf{x}, \mathbf{w}_1, \dots, \mathbf{w}_L) = h^L(h^{L-1}(\dots h^1(\mathbf{x}, \mathbf{w}_1), \dots, \mathbf{w}_{L-1}), \mathbf{w}_L)$$



Ex: пример модели сети: многослойный перцептрон с двумя скрытыми слоями

- Прямое распространение сигналов по модели (в частности: нейронной сети) реализуется с помощью **прямого прохода (forward pass)**: входящая информация (вектор \mathbf{x}) распространяется через сеть f_L с учетом весов связей $\boldsymbol{\theta}$, рассчитывается выходной вектор $\hat{\mathbf{y}} = f_L(\mathbf{x}, \boldsymbol{\theta})$.
 - Каждый слой нейронной сети - это последовательно применяемая функция слоя, которая рассчитывается при помощи операций с тензорами.

$$\bar{h} = f_h(W_1^T \bar{x})$$



14

Ex: пример прямого прохода

In [158]:

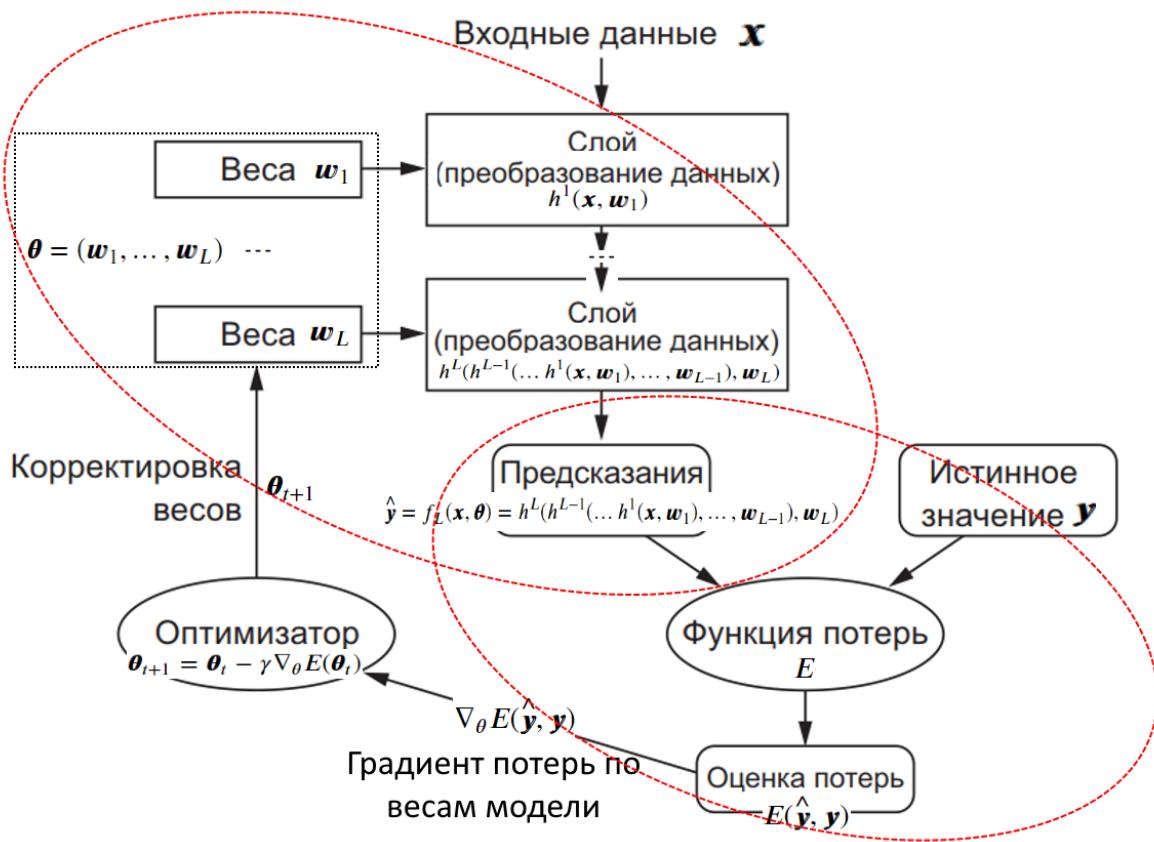
```
# Почему Тензор?
# Персептрон на тензорах
# Два слоя перспетрнов
```

- Для решения этой задачи и используется алгоритм обратного распространения ошибки (backpropagation).

In [86]:

```
# реализация на PyTorch
# Линейная регрессия
# Персептрон
```

- Последовательность операций с тензорами используется для расчета результата:
 - Прямой проход (forward pass): входящая информация (вектор \mathbf{x}) распространяется через сеть с учетом весов связей, рассчитывается выходной вектор $\hat{\mathbf{y}} = f_L(\mathbf{x}, \boldsymbol{\theta}) = h^L(h^{L-1}(\dots h^1(\mathbf{x}, \mathbf{w}_1), \dots, \mathbf{w}_{L-1}), \mathbf{w}_L)$
 - Оценки ошибки $E(\hat{\mathbf{y}}, \mathbf{y})$ на множестве правильных ответов: \mathbf{y} .



Прямой проход и оценка ошибки

In [3]:

```
import torch
```

In [86]:

```
# Модель линейной регрессии (с несколькими параметрами)
#  $f = X * w$ 

# Данные для обучения:
# признаки X: рассматривается 4 наблюдения (ось 0) и 2 признака (ось 1):

# Вариант исходных данны #1 (2й признак всегда равен 0):
# X = torch.tensor([[1., 0.],
#                   [2., 0.],
#                   [3., 0.],
#                   [4., 0.]], dtype=torch.float32) # Size([4, 2])

# Вариант исходных данны #2 (2й признак используется и существенно больше 1го):
X = torch.tensor([[1., 40.],
                  [2., 30.],
                  [3., 20.],
                  [4., 10.]], dtype=torch.float32) # Size([4, 2])

print(f'Матрица X: \n{X}')
print(f'X.size = {X.size()}')

# истинное значение весов (используется только для получения обучающих правильных ответов):

# Вариант #1:
# w_ans = torch.tensor([2., 0.], dtype=torch.float32)

# Вариант #2:
w_ans = torch.tensor([2., 1.], dtype=torch.float32)

# Y - правильные ответы:
Y = X @ w_ans

torch.set_printoptions(precision=5) # точность вывода на печать значений тензоров
print(f'w true value = {w_ans}, Y = {Y}')
```

Матрица X:

```
tensor([[ 1., 40.],
       [ 2., 30.],
       [ 3., 20.],
       [ 4., 10.]])
X.size = torch.Size([4, 2])
w true value = tensor([2., 1.]), Y = tensor([42., 34., 26., 18.])
```

In [87]:

```
# model (модель, в нашем случае: линейная регрессия)

# изначальное значение весов w
w = torch.tensor([0.0, 0.0], dtype=torch.float32, requires_grad=False)

print(f'w:{\n{w}\n}')

# прямое распространение:
def forward(X):
    return X @ w # Size([4])

# Loss = MSE (функция потерь, в нашем случае: средняя квадратичная ошибка)
def loss(y, y_pred):
    return ((y_pred - y)**2).mean() # Size([])

# градиент:
# рассчитан аналитически по модели и функции потерь:
# J = MSE = 1/N * (w*x - y)**2
# dJ/dw = 1/N * 2 * (w*x - y) * x
def gradient(x, y, y_pred):
    #     print(f'''y = {y},
    #     y_pred = {y_pred},
    #     (2* (y_pred - y)).unsqueeze(1) = {(2* (y_pred - y)).unsqueeze(1)},
    #     x = {x},
    #     ((2* (y_pred - y)).unsqueeze(1) * x) = {((2* (y_pred - y)).unsqueeze(1) * x)},
    #     ((2* (y_pred - y)).unsqueeze(1) * x).mean(dim=0) = {((2* (y_pred - y)).unsqueeze(1) *
    return ((2* (y_pred - y)).unsqueeze(1) * x).mean(dim=0)
```

w:
tensor([0., 0.])

In [88]:

```
# Training

# Вариант #1:
# learning_rate = 0.05
# n_iters = 20 + 1

# Вариант #2:
learning_rate = 0.0013
n_iters = 1000 + 1

# основной цикл:
for epoch in range(n_iters):
    # predict = forward pass
    y_pred = forward(X)

    # Loss
    l = loss(Y, y_pred)

    # calculate gradients
    dw = gradient(X, Y, y_pred)

    # update weights
    w -= learning_rate * dw

    if epoch % 5 == 0:
        print(f'epoch {epoch}: w = {w}, y_pred = {y_pred}, loss = {l:.8f}\ngradient = {dw}'
```

```
epoch 0: w = tensor([0.16900, 2.21000]), y_pred = tensor([0., 0., 0.,
0.]), loss = 980.00000000
gradient = tensor([-130., -1700.])
epoch 5: w = tensor([0.13813, 0.24422]), y_pred = tensor([81.70274, 61.575
23, 41.44772, 21.32021]), loss = 646.58898926
gradient = tensor([ 77.23860, 1378.76135])
epoch 10: w = tensor([0.33966, 1.82453]), y_pred = tensor([15.22918, 11.70
162, 8.17406, 4.64650]), loss = 427.49359131
gradient = tensor([-89.12970, -1114.91895])
epoch 15: w = tensor([0.34364, 0.53338]), y_pred = tensor([68.78387, 52.09
386, 35.40386, 18.71385]), loss = 283.42648315
gradient = tensor([ 47.01929, 904.69324])
epoch 20: w = tensor([0.49880, 1.56850]), y_pred = tensor([25.13912, 19.37
721, 13.61530, 7.85340]), loss = 188.61242676
gradient = tensor([-61.92349, -731.13959])
epoch 25: w = tensor([0.52316, 0.72007]), y_pred = tensor([60.23328, 45.87
371, 31.51414, 17.15457]), loss = 126.13953400
gradient = tensor([ 27.57071, 593.68567])
epoch 30: w = tensor([0.64554, 1.39771]), y_pred = tensor([31.56791, 24.41
101.175570, 10.00001, 1.01.0100000])
```

--
Обучение модели нейронной сети, алгоритм обратного распространения ошибки [
* \[\[к оглавлению\]\(#\)\] \(#разделы\)](#)

Разделы: [
* \[\[Базовые понятия, персептрон\]\(#\)\] \(#персептрон\)
* \[\[Современные методы обучения нейронной сети и обратное распространение ошибки\]\(#\)\].
\(#современные-методы\)
*](#)

- * [\[Обучение модели нейронной сети, алгоритм обратного распространения ошибки\].\(#обратное-распространение\)](#)
 - * [\[Вторая весна ИИ и глубокое обучение\].\(#вторая-весна\)](#)
 - * [\[Вторая весна ИИ и глубокое обучение\].\(#вторая-весна\)](#)
 - * [\[Вторая весна ИИ и глубокое обучение\].\(#вторая-весна\)](#)
-
- * [\[к оглавлению\].\(#разделы\)](#)

In []:

```
# почему эта ячейка тут??
```

In [4]:

```
# requires_grad argument
# This will tell pytorch that it will need to calculate the gradients for this tensor
# Later in your optimization steps
# i.e. this is a variable in your model that you want to optimize
x = torch.tensor([5.5, 3], requires_grad=True)
```

In []:

Проблема обучения модели нейронной сети

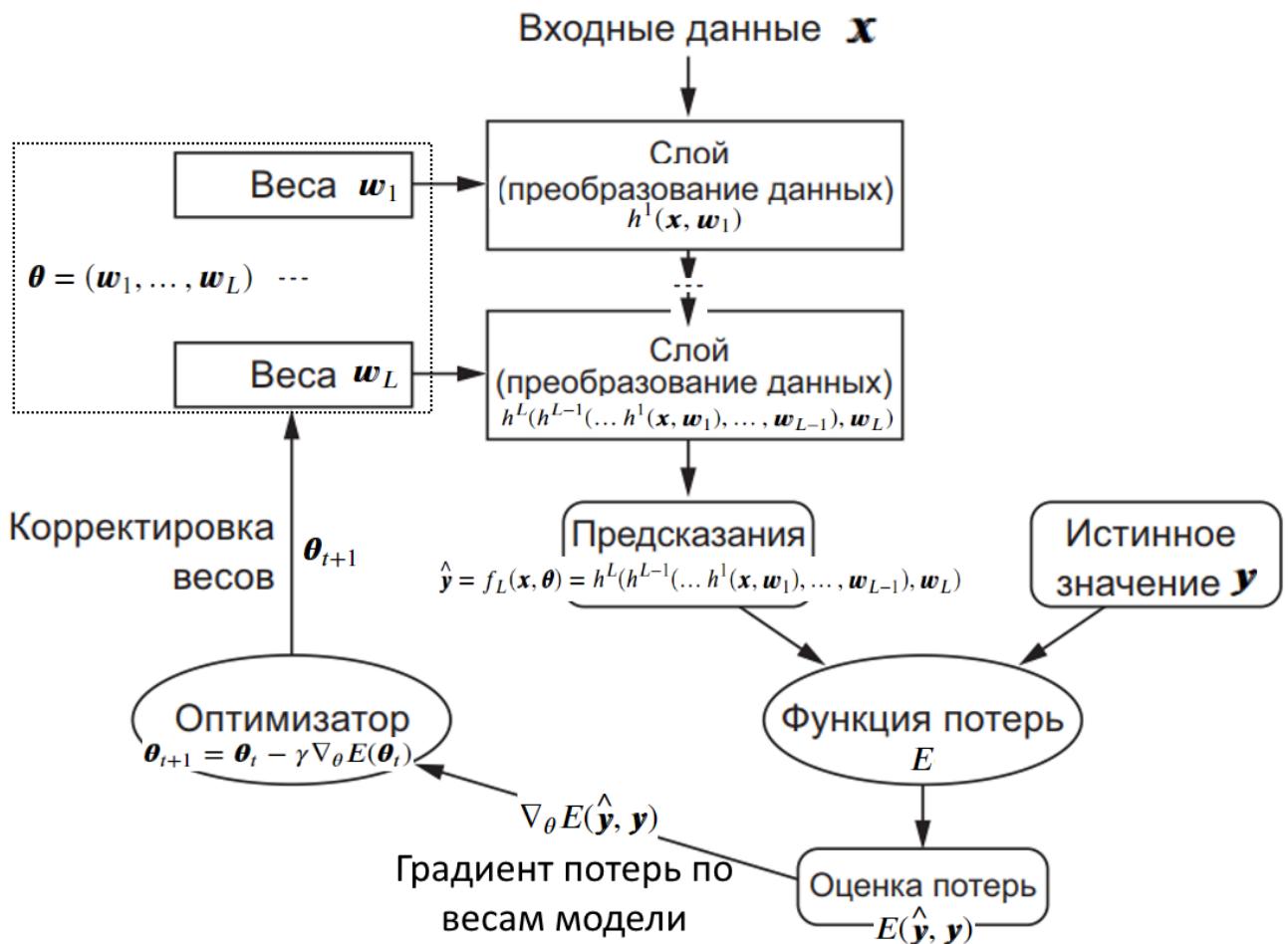
- [к оглавлению](#)

Проблема обучения модели нейронной сети: общий взгляд

- **Note:** основная проблема это не применение модели к входным данным \mathbf{x} и оценка ошибки на правильных ответах \mathbf{y} , а **обучение модели** (определение наилучших параметров модели $\boldsymbol{\theta}$).
 - В случае нейронной сети обучение сводится к поиску весов слоев сети $\boldsymbol{\theta} = (\mathbf{w}_1, \dots, \mathbf{w}_L)$, которые в совокупности являются параметрами модели $\boldsymbol{\theta}$.
- Формально: цель обучения - найти оптимальное значение параметров θ^* , минимизирующих ошибку на обучающей выборке D :

$$\theta^* = \arg \min_{\boldsymbol{\theta}} E(\boldsymbol{\theta}) = \arg \min_{\boldsymbol{\theta}} \sum_{(\mathbf{x}, \mathbf{y}) \in D} E(f_L(\mathbf{x}, \boldsymbol{\theta}), \mathbf{y})$$

- Т.е. задача обучения сводится к задаче оптимизации.
 - **Note:** На самом деле **все сложнее**: хороший результат на D может плохо обобщаться (модель может давать низкое качество на другой выборке из той же генеральной совокупности) - **проблема переобучения**.



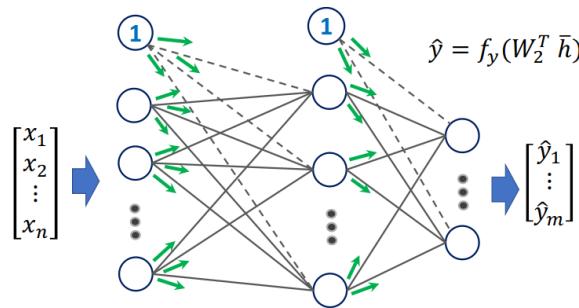
Принципиальная логика обучения нейронной сети

Прямой проход и оценка ошибки

- **Прямой проход** (forward pass): входящая информация (вектор \mathbf{x}) распространяется через сеть с учетом весов связей, рассчитывается выходной вектор

$$\hat{\mathbf{y}} = f_L(\mathbf{x}, \theta) = h^L(h^{L-1}(\dots h^1(\mathbf{x}, \mathbf{w}_1), \dots, \mathbf{w}_{L-1}), \mathbf{w}_L)$$

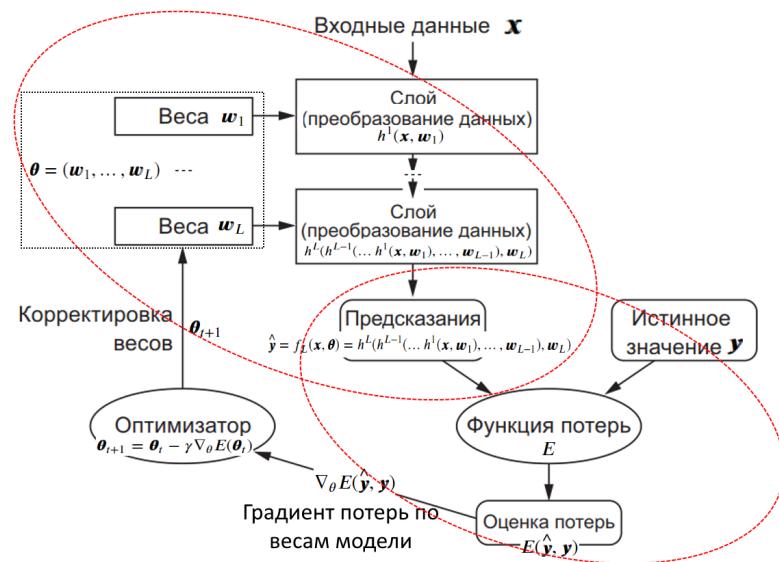
$$\bar{h} = f_h(W_1^T \bar{x})$$



14

Ex: пример прямого прохода

- Оценки ошибки $E(\hat{\mathbf{y}}, \mathbf{y})$ на множестве правильных ответов: \mathbf{y} .



Прямой проход и оценка ошибки в общей логике обучения нейронной сети

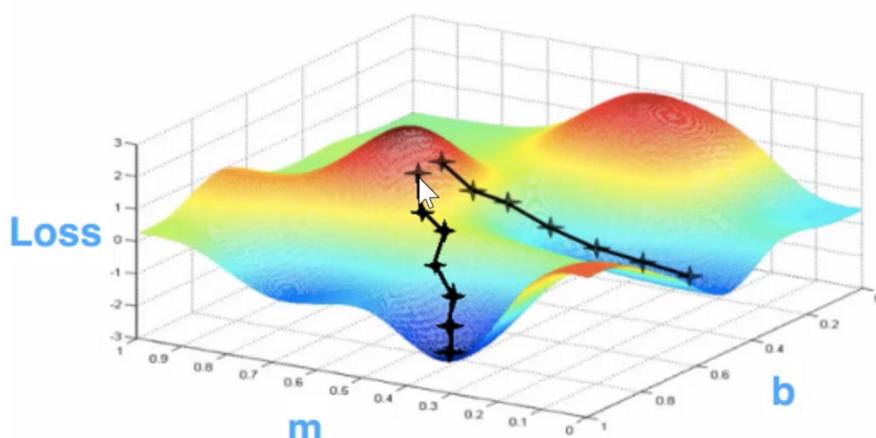
Задача оптимизации

- Задача: корректировка весов сети (параметров модели θ) на основе информации об ошибке на обучающих примерах $E(\hat{\mathbf{y}}, \mathbf{y})$.
 - Решение: использовать методы оптимизации, основанные на **методе градиентного спуска**.
- Метод градиентного спуска** - метод нахождения локального экстремума (минимума или максимума) функции с помощью движения вдоль градиента. В нашем случае шаг метода градиентного спуска выглядит следующим образом:

$$\theta_t = \theta_{t-1} - \gamma \nabla_{\theta} E(\theta_{t-1}) = \theta_{t-1} - \gamma \sum_{(\mathbf{x}, \mathbf{y}) \in D} \nabla_{\theta} E(f_L(\mathbf{x}, \theta), \mathbf{y})$$

- Note:** Выполнение на каждом шаге градиентного спуска суммирование по всем $(\mathbf{x}, \mathbf{y}) \in D$ **обычно слишком неэффективно**
- Для выпуклых функций **задача локальной оптимизации** - найти локальный минимум (максимум) автоматически превращается в **задачу глобальной оптимизации** - найти точку, в которой достигается наименьшее (наибольшее) значение функции, то есть самую низкую (высокую) точку среди всех.
- Оптимизировать веса одного перцептрона - выпуклая задача, но **для большой нейронной сети целевая функция не является выпуклой**.

$$f(x) = \text{nonlinear function of } x$$

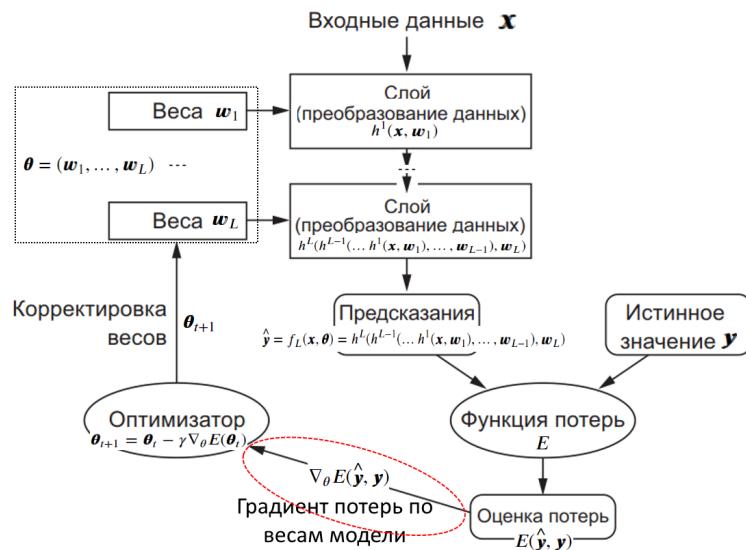


Пример работы градиентного спуска для функции двух переменных

- У нейронных сетей функция ошибки может задавать **очень сложный ландшафт** с огромным числом локальных максимумов и минимумов. Это свойство необходимо для обеспечения **выразительности нейронных сетей**, позволяющей им решать так много разных задач.
- Note:** для использования методов, основанных на методе градиентного спуска **необходимо знать градиент функции потерь по параметрам модели**: $\nabla_{\theta} E(f_L(\mathbf{x}, \boldsymbol{\theta}), \mathbf{y})$. Этот градиент определяет вектор ("направление") изменения параметров.

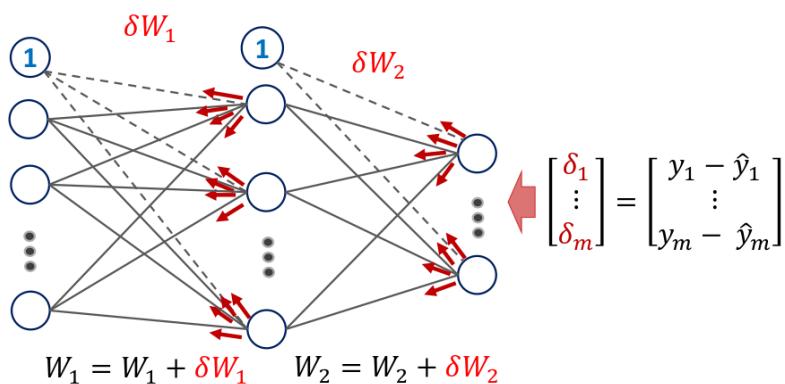
Проблема поиска градиента

- Q:** Проблема: как найти градиент для нейронной сети: $\nabla_{\theta} E(f_L(\mathbf{x}, \boldsymbol{\theta}), \mathbf{y})$?



Проблема поиска градиента в общей логике обучения нейронной сети

- Для решения этой задачи и используется **алгоритм обратного распространения ошибки** (backpropagation). Суть алгоритма:
 - рассчитывается ошибка между выходным вектором сети $\hat{\mathbf{y}}$ и правильным ответом обучающего примера \mathbf{y}
 - ошибка распространяется от результата к источнику (в обратную сторону) для корректировки весов

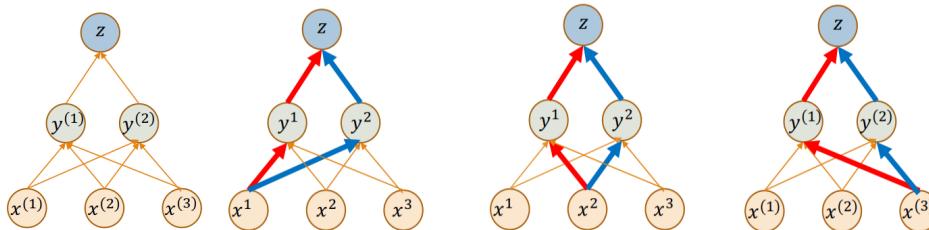


Ex: Пример обратного распространения ошибки

Рассчет градиента суперпозиции двух функций нескольких переменных

- Сначала рассмотрим подзадачу: как рассчитать градиент для $f_L(\mathbf{x}, \mathbf{w}_1, \dots, \mathbf{w}_L) = h^L(h^{L-1}(\dots h^1(\mathbf{x}, \mathbf{w}_1), \dots, \mathbf{w}_{L-1}), \mathbf{w}_L)$?
- Для этого нам нужно будет **рассчитывать градиент суперпозиции (сложной функции)** состоящей из последовательного применения функций слоев h^i .
- Вспомним, как рассчитать производную (градиент) суперпозиции нескольких функций.
 - Пусть $z = f(y)$, $y = g(x)$
 - Тогда производная суперпозиции функций (правило дифференцирования сложной функции (chain rule)): $\frac{dz}{dx} = \frac{dz}{dy} \frac{dy}{dx}$
 - Если $\mathbf{x} \in \mathbb{R}^n$, $\mathbf{y} \in \mathbb{R}^m$, а $z \in \mathbb{R}$, то: $\frac{\partial z}{\partial x_i} = \sum_j \frac{\partial z}{\partial y_j} \frac{\partial y_j}{\partial x_i}$

Примеры расчета градиента суперпозиции двух функций нескольких переменных:



$$\frac{dz}{dx^1} = \frac{dz}{dy^1} \frac{dy^1}{dx^1} + \frac{dz}{dy^2} \frac{dy^2}{dx^1} \quad \frac{dz}{dx^2} = \frac{dz}{dy^1} \frac{dy^1}{dx^2} + \frac{dz}{dy^2} \frac{dy^2}{dx^2} \quad \frac{dz}{dx^3} = \frac{dz}{dy^1} \frac{dy^1}{dx^3} + \frac{dz}{dy^2} \frac{dy^2}{dx^3}$$

Т.е. нам нужны градиенты по всем возможным путям (рассмотренным в обратном порядке) зависимостей переменных.

Запись этой же задачи в векторной нотации:

$$\bullet \frac{dz}{dx} = \nabla_x(z) = \begin{pmatrix} \frac{\partial z}{\partial x_1} \\ \vdots \\ \frac{\partial z}{\partial x_n} \end{pmatrix} = \left(\frac{dy}{dx} \right)^T \cdot \nabla_y(z) = J(\mathbf{y}(\mathbf{x}))^T \cdot \nabla_y(z) = J(\mathbf{y}(\mathbf{x}))^T \cdot \begin{pmatrix} \frac{\partial z}{\partial y_1} \\ \vdots \\ \frac{\partial z}{\partial y_m} \end{pmatrix}$$

• Где J это Якобиан:

$$J(\mathbf{y}(\mathbf{x})) = \begin{pmatrix} \frac{\partial y_1}{\partial x_1} & \dots & \frac{\partial y_1}{\partial x_n} \\ \vdots & \ddots & \vdots \\ \frac{\partial y_m}{\partial x_1} & \dots & \frac{\partial y_m}{\partial x_n} \end{pmatrix}$$

Задача поиска градиента: $\nabla_{\theta} E(f_L(\mathbf{x}, \theta), \mathbf{y})$

- Перейдем от f_L к последовательному расчету функций слоев h^i :
 $\nabla_{\theta} E(f_L(\mathbf{x}, \theta), \mathbf{y}) = \nabla_{\mathbf{w}_i} E(f_L(\mathbf{x}, \mathbf{w}_1, \dots, \mathbf{w}_L), \mathbf{y}) = \nabla_{\mathbf{w}_i} E(h^L(h^{L-1}(\dots h^1(\mathbf{x}, \mathbf{w}_1), \dots, \mathbf{w}_{L-1}), \mathbf{w}_L), \mathbf{y})$
- Обозначим через \mathbf{a}^l результат расчета функции активации на слое l : $\mathbf{a}^l = h^l(\mathbf{x}_l, \mathbf{w}_l)$. Тогда:
 $\mathbf{x}_{l+1} = \mathbf{a}^l$ (вход следующего слоя является результатом расчета функции активации предыдущего слоя)
- Тогда можно записать: $\nabla_{\theta} E(f_L(\mathbf{x}, \theta), \mathbf{y}) = \nabla_{\theta} E(\mathbf{a}^L, \mathbf{y})$. Функция потерь $E(\mathbf{a}^L, \mathbf{y})$ зависит от \mathbf{a}^L , \mathbf{a}^L от $\mathbf{a}^{L-1}, \dots, \mathbf{a}^{l+1}$ от \mathbf{a}^l
- Исходя из этого представления можно градиенты явесов l -го слоя можно записать как:

$$\frac{\partial E}{\partial \mathbf{w}_l} = \frac{\partial E}{\partial \mathbf{a}_L} \cdot \frac{\partial \mathbf{a}_L}{\partial \mathbf{a}_{L-1}} \cdot \dots \cdot \frac{\partial \mathbf{a}_{l+1}}{\partial \mathbf{a}_l} \cdot \frac{\partial \mathbf{a}_l}{\partial \mathbf{w}_l}$$

- Произведение всех сомножителей кроме последнего является градиентом функции потерь по результатам расчета функции активации слоя l :

$$\frac{\partial E}{\partial \mathbf{a}_L} \cdot \frac{\partial \mathbf{a}_L}{\partial \mathbf{a}_{L-1}} \cdot \dots \cdot \frac{\partial \mathbf{a}_{l+1}}{\partial \mathbf{a}_l} = \frac{\partial E}{\partial \mathbf{a}_l}$$

- Тогда:

$$\frac{\partial E}{\partial \mathbf{w}_l} = \left(\frac{\partial \mathbf{a}_l}{\partial \mathbf{w}_l} \right)^T \cdot \frac{\partial E}{\partial \mathbf{a}_l}$$

для расчета $\frac{\partial \mathbf{a}_l}{\partial \mathbf{w}_l}$ нам нужен только якобиан функции активации l -го слоя по параметрам слоя \mathbf{w}_l .

- Градиент функции потерь по результатам расчета функции активации слоя l может быть рассчитан рекурсивно по результатам слоя l , собственно тут и происходит **обратное распространение**:

$$\frac{\partial E}{\partial \mathbf{a}_l} = \left(\frac{\partial \mathbf{a}_{l+1}}{\partial \mathbf{a}_l} \right)^T \cdot \frac{\partial E}{\partial \mathbf{a}_{l+1}} = \left(\frac{\partial \mathbf{a}_{l+1}}{\partial \mathbf{x}_{l+1}} \right)^T \cdot \frac{\partial E}{\partial \mathbf{a}_{l+1}}$$

для расчета $\frac{\partial \mathbf{a}_{l+1}}{\partial \mathbf{x}_{l+1}}$ нам нужен только якобиан функции активации $l+1$ -го слоя по входным значениям слоя \mathbf{x}_{l+1}

- Т.е. чтобы проводить обратное распространение ошибки, нам на каждом слое (например l -м) нужно рассчитывать два якобиана:

- якобиан функции активации l -го слоя по параметрам слоя $\frac{\partial \mathbf{a}_l}{\partial \mathbf{w}_l}$ - он позволит рассчитать

градиент $\frac{\partial E}{\partial \mathbf{w}_l}$ и сделать очередной шаг градиентного спуска для параметров этого слоя:

$$\mathbf{w}_l^{t+1} = \mathbf{w}_l^t - \gamma \nabla_{\mathbf{w}_l} E(\mathbf{w}^t) = \mathbf{w}_l^t - \gamma \frac{\partial E(\mathbf{w}^t)}{\partial \mathbf{w}_l}$$

- якобиан функции активации l -го слоя по входным значениям слоя: $\frac{\partial \mathbf{a}_l}{\partial \mathbf{x}_l}$ - он позволит распространить ошибку на низлежащие слои.

Таким образом при обучении нам нужна только **очень локальная информация, содержащаяся в самом слое**. Т.е.:

- нет необходимости знать как устроены соседние слои:** между слоями **очень простой интерфейс**
- т.е. **можно создать модульную архитектуру для слоев нейронной сети:** каждый модуль рассчитывает значение функции активации на основе выходов на прямом проходе и распространяет ошибку пришедшую на выходы на обратном проходе; все модули стандартным образом стыкуются друг с другом
- при модульной архитектуре граф нейронной сети может быть очень сложным, но его **рассчет выполняется по одной простой и универсальной схеме**
- внутри модули могут быть сложно устроены, это никак не меняет логику остальных модулей и всего процесса обучения**, главное чтобы модуль корректно выполнял прямой и обратный проход.

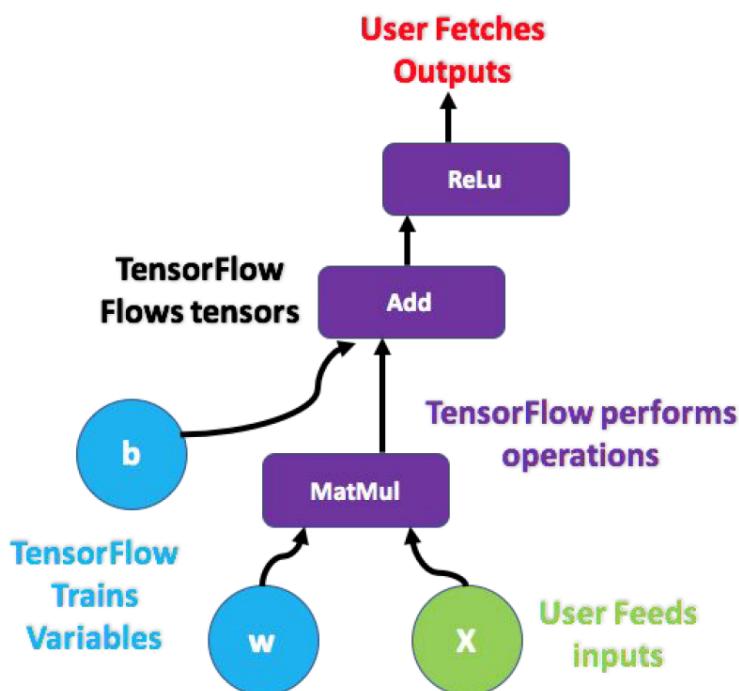
Дифференцируемое программирование и реализация обратного распространения ошибки

Почему **Tensor Flow?**

Q: Как реализовать **алгоритм обратного распространения ошибки** удобно для использования в задачах моделирования ИНС?

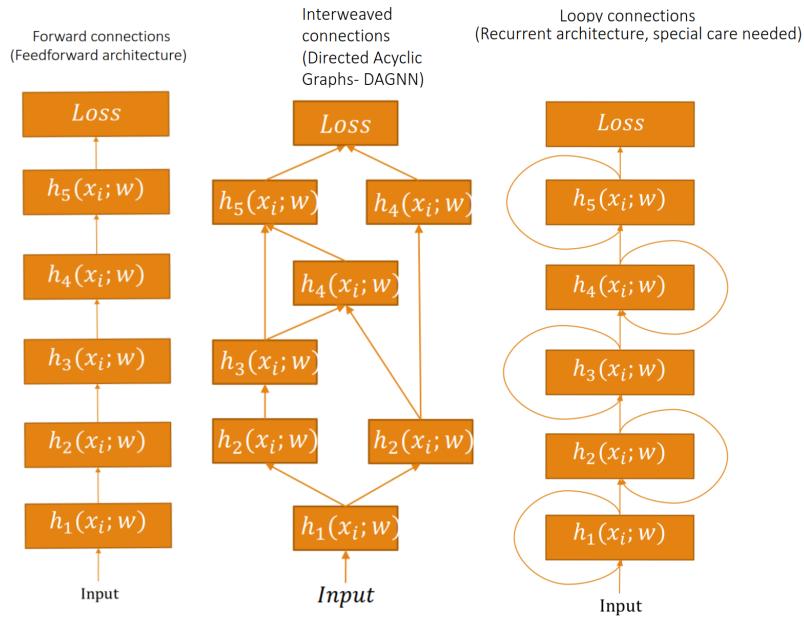
Основная абстракция TensorFlow, PyTorch и других аналогичных библиотеках - **граф потока вычислений**.

- Рассматриваемые библиотеки обычно:
 1. задают **граф потока вычислений** (формирует объект отложенных вычислений)
 2. запускают **процедуру выполнения отложенных вычислений** и получает **результаты вычислений** (в т.ч. ошибку модели).
- Возможность в явном виде работать с графиком потока вычислений дает большое преимущество для **автоматического решения задачи обратного распространения ошибки**, являющейся составляющей задачей обучения модели ИНС.



Принцип устройства графа потока вычислений в TensorFlow

- Нейронная сеть это иерархия (она может быть простой и очень сложной) связанных (последовательно применяемых) функций слоев ИНС. Модель сети f_L может быть представлена как суперпозиция из L слоев h^i , $i \in \{1, \dots, L\}$, каждый из которых параметризуется своими весами w_i :
$$f_L(\mathbf{x}, \boldsymbol{\theta}) = f_L(\mathbf{x}, \mathbf{w}_1, \dots, \mathbf{w}_L) = h^L(h^{L-1}(\dots h^1(\mathbf{x}, \mathbf{w}_1), \dots, \mathbf{w}_{L-1}), \mathbf{w}_L)$$
- Вычисление функций слоев и взаимосвязи между слоями формируют график потока вычислений в библиотеке моделирования ИНС.



Примеры иерархий в нейронных сетях

- По сути, ИНС это композиция модулей, представляющих собой слои нейронной сети:
 - если сеть прямого распространения (feedforward), то все просто
 - если сеть является направленным ациклическим графом, то существует правильный порядок применения функций
 - в случае, если есть циклы, образующие рекуррентные связи, то существуют специальные подходы (будут рассмотрены позднее)
- На обратном проходе (при обратном распространении ошибки) нам необходимо **дифференцировать сложную функцию** многослойной ИНС

$$\nabla_{\theta} E(f_L(\mathbf{x}, \theta), \mathbf{y}) = \nabla_{\mathbf{w}_i} E(f_L(\mathbf{x}, \mathbf{w}_1, \dots, \mathbf{w}_L), \mathbf{y}) = \nabla_{\mathbf{w}_i} E(h^L(h^{L-1}(\dots h^1(\mathbf{x}, \mathbf{w}_1), \dots, \mathbf{w}_{L-1}), \mathbf{w}_L), \mathbf{y})$$
- алгоритм обратного распространения ошибки позволяет свести эту задачу к дифференцированию составляющих функций, но для этого необходимо **хранить информацию о виде и взаимосвязях функций задействованных в расчете модели ИНС**, именно эта информация и хранится в графе потока вычислений. Это позволяет организовать **автоматическое дифференцирование** сложной функции многослойной ИНС.

Дифференцируемое программирование

Def: **Дифференцируемое программирование** (differentiable programming) - парадигма программирования при которой программа (функция расчета значения) может быть продифференцирована в любой точке, обычно с помощью **автоматического дифференцирования**.

Это свойство позволяет использовать к программе **методы оптимизации основанные на расчете градиента**, обычно - **методы градиентного спуска**.

Дифференцируемое программирование используется в:

- глубоком обучении
- глубоком обучении комбинированном с физическими моделями в робототехнике
- специализированных методах трассировки лучей
- обработке изображений

Большинство фреймворков для дифференцируемого программирования использует граф потока вычислений определяющий выполнение программы и ее структуры данных.

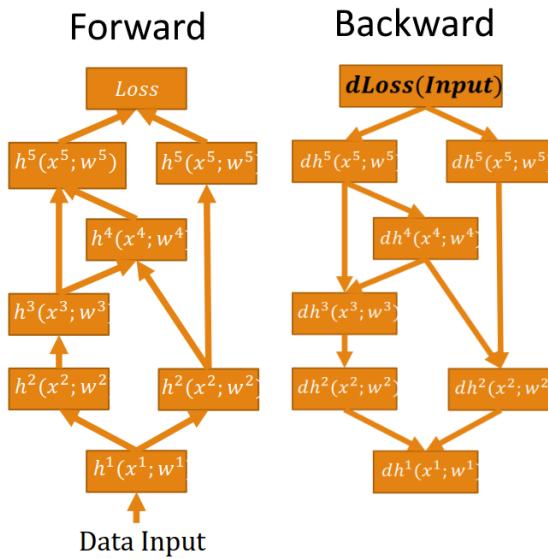
Основные классы фреймворков для дифференцируемого программирования:

- **статические** - они компилируют граф потока вычислений. Типичные представители: TensorFlow, Theano и др. Плюсы и минусы
 - + могут использовать оптимизацию при компиляции
 - + легче масштабируются на большие системы
 - - статичность ограничивает интерактивность
 - - многие программы не могут реализовываться легко (в частности: циклы, рекурсия)
- **динамические** - динамически исполняют граф потока вычислений. Используют перегрузку операторов для записи. Типичные представители: PyTorch, AutoGradFlow. Плюсы и минусы:
 - + более простая и понятная запись программы
 - - накладные расходы интерпретатора
 - - невозможно использовать оптимизацию компилятора
 - - хуже масштабируемость
- статическая на основе разбора промежуточного представления синтаксического разбора исходной программы. Пример фреймворк Zygote (язык программирования Julia).

Прямой проход:

- Модули из графа обходятся один за одним начиная с узла входных данных и далее по мере готовности всех необходимых входных данных для очередного модуля, который еще не был обойден
- Рассчет функций активации для каждого модуля по входным данным: $a_l = h_l(x_l, w_l)$
- Промежуточные значения кэшируются, чтобы не рассчитывать их повторно (в сложном графе сети и при обратном проходе)
- Выходы одних модулей становятся входами других модулей: $x_{l+1} = a_l$
- Последним модулем рассчитывается сумма потерь для входных данных

Прямой и обратный проход процедуры обучения многослойной ИНС:



Обратный проход:

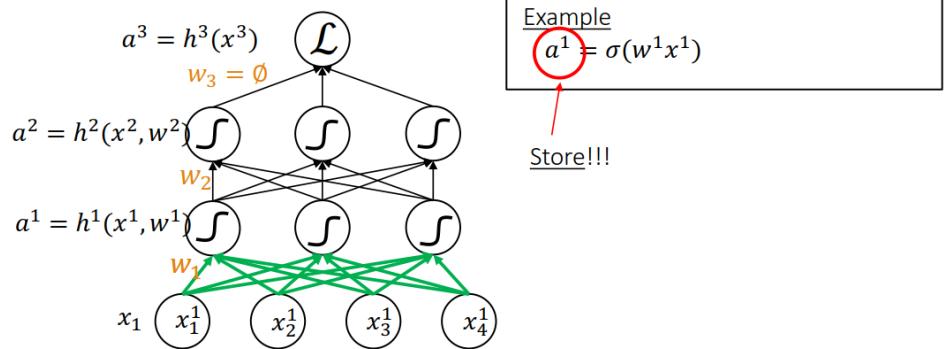
- Сначала должен быть произведен прямой проход. На входе обратного прохода известна сумма потерь.
- Строится обратный порядок обхода графа зависимостей модулей.
- Модули из графа обходятся один за одним начиная с узла расчета функции потерь и далее по мере готовности всех необходимых входных данных для очередного модуля, который еще не был обойден

- Для каждого модуля рассчитывается якобиан функции активации по параметрам слоя $\frac{\partial \mathbf{a}_l}{\partial \mathbf{w}_l}$ и якобиан функции активации по входным значениям слоя: $\frac{\partial \mathbf{a}_l}{\partial \mathbf{x}_l}$
- По пришедшему в модуль градиенту ошибки (полученному из модулей использовавших результаты данного модуля на прямом проходе) $\frac{\partial E}{\partial \mathbf{a}_l}$ рассчитывается:
 - Градиент для шага градиентного спуска по параметрам модуля w_l : $\frac{\partial E}{\partial \mathbf{w}_l} = \left(\frac{\partial \mathbf{a}_l}{\partial \mathbf{w}_l} \right)^T \cdot \frac{\partial E}{\partial \mathbf{a}_l}$
 - Градиент ошибки, который передается в модули, поставившие данные в этот модуль во время прямого прохода: $\frac{\partial E}{\partial \mathbf{a}_{l-1}} = \left(\frac{\partial \mathbf{a}_l}{\partial \mathbf{x}_l} \right)^T \cdot \frac{\partial E}{\partial \mathbf{a}_l}$

Пример: прямой проход, шаг 1

Forward propagations

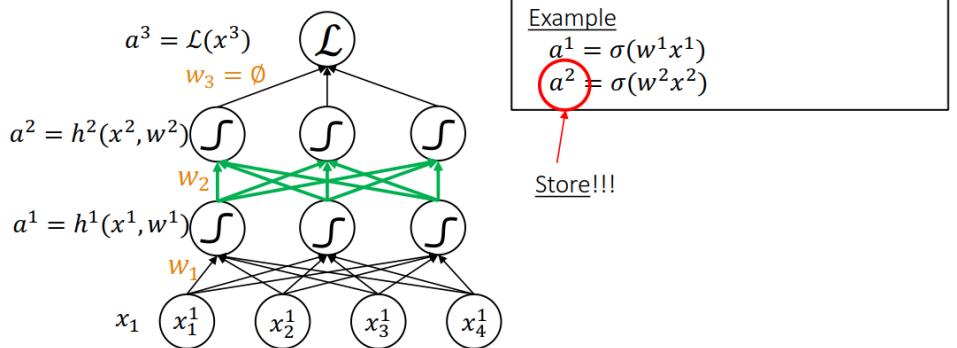
Compute and store $a_1 = h_1(x_1)$



Пример: прямой проход, шаг 2

Forward propagations

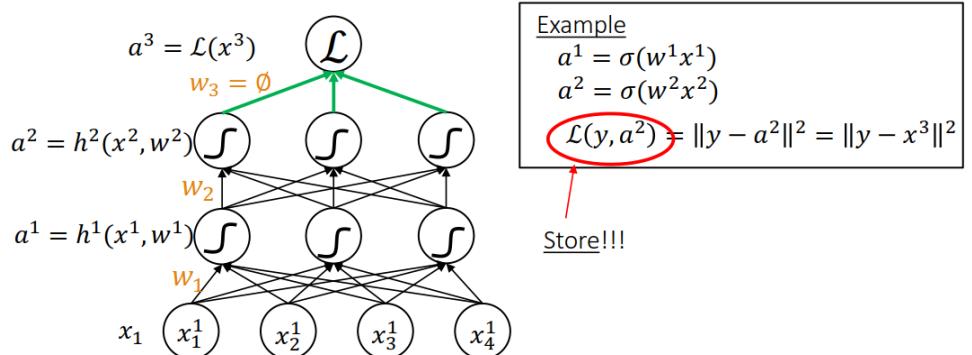
Compute and store $a_2 = h_2(x_2)$



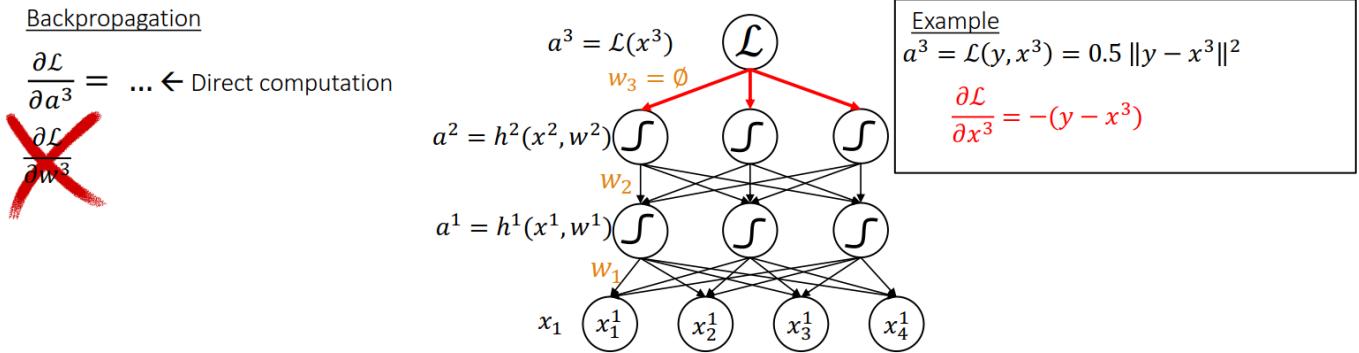
Пример: прямой проход, шаг 3

Forward propagations

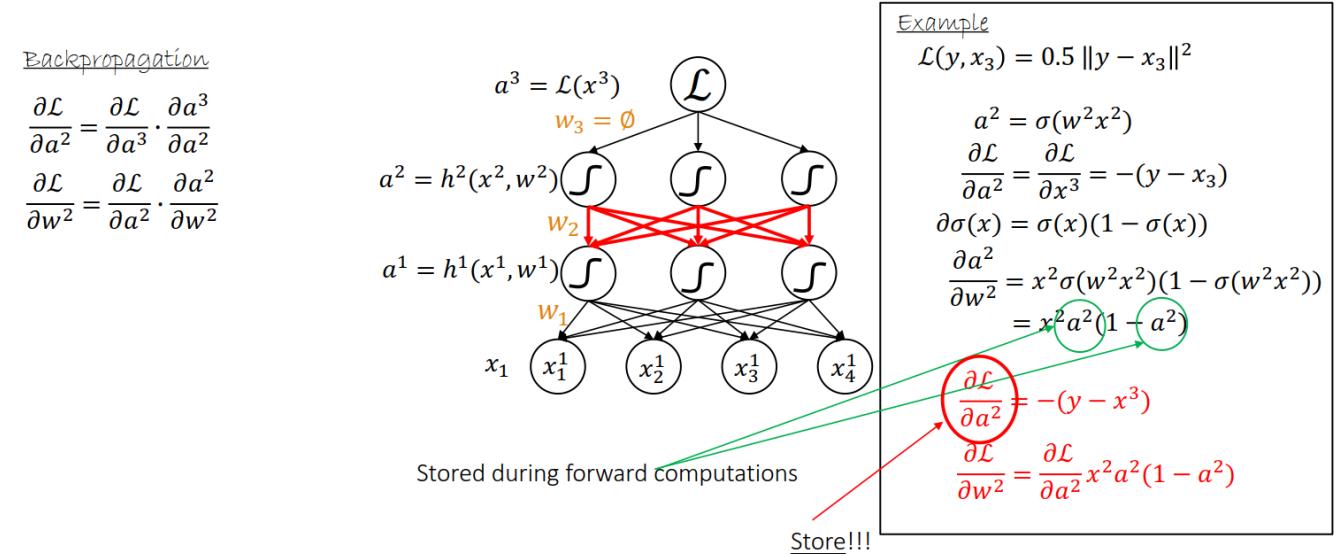
Compute and store $a_3 = h_3(x_3)$



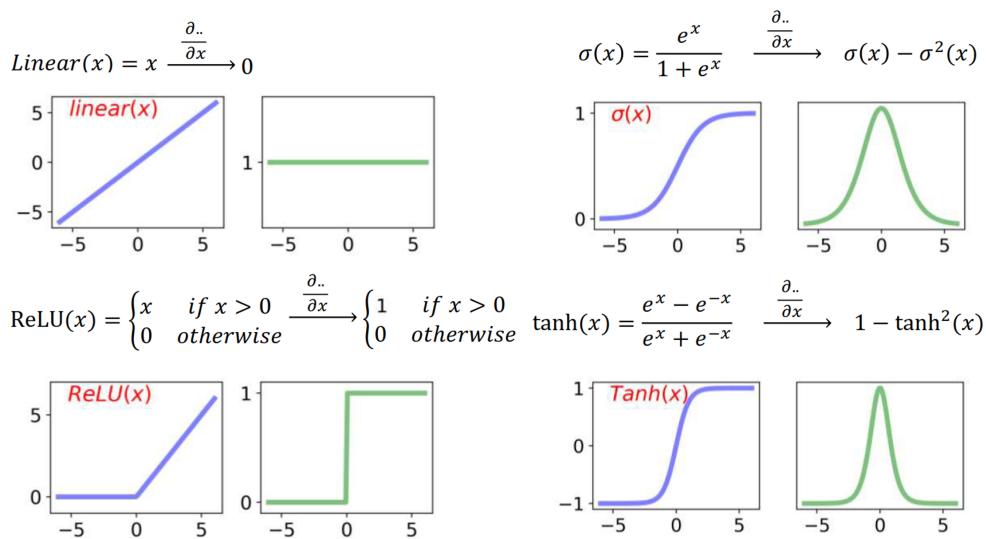
Пример: обратный проход, шаг 1



Пример: обратный проход, шаг 2



Производные популярных функций активации

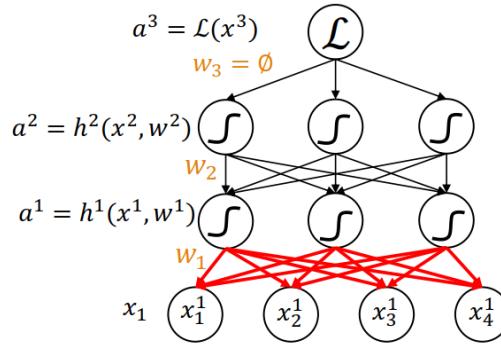


Пример: обратный проход, шаг 3

Backpropagation

$$\frac{\partial \mathcal{L}}{\partial a_1} = \frac{\partial \mathcal{L}}{\partial a_2} \cdot \frac{\partial a_2}{\partial a_1}$$

$$\frac{\partial \mathcal{L}}{\partial w_1} = \frac{\partial \mathcal{L}}{\partial a_1} \cdot \frac{\partial a_1}{\partial w_1}$$



Computed from the exact previous
backpropagation step (Remember, recursive rule)

Example

$$\mathcal{L}(y, a^3) = 0.5 \|y - a^3\|^2$$

$$a^2 = \sigma(w^2 x^2)$$

$$a^1 = \sigma(w^1 x^1)$$

$$\frac{\partial a^2}{\partial a^1} = \frac{\partial a^2}{\partial a^2} = w^2 a^2 (1 - a^2)$$

$$\frac{\partial a^1}{\partial w^1} = x^1 a^1 (1 - a^1)$$

$$\frac{\partial \mathcal{L}}{\partial a^1} = \frac{\partial \mathcal{L}}{\partial a^2} w^2 a^2 (1 - a^2)$$

$$\frac{\partial \mathcal{L}}{\partial w^1} = \frac{\partial \mathcal{L}}{\partial a^1} x^1 a^1 (1 - a^1)$$

Автоматическое дифференцирование в PyTorch:

In [90]:

```
# The autograd package provides automatic differentiation
# for all operations on Tensors

# requires_grad = True -> tracks all operations on the tensor.
x = torch.randn(3, requires_grad=True)
y = x + 2

# y was created as a result of an operation, so it has a grad_fn attribute.
# grad_fn: references a Function that has created the Tensor
print(x) # created by the user -> grad_fn is None
print(y)
print(y.grad_fn)
```

```
tensor([ 0.21713,  1.01658, -0.62985], requires_grad=True)
tensor([2.21713, 3.01658, 1.37015], grad_fn=<AddBackward0>)
<AddBackward0 object at 0x00000238229A9408>
```

In [92]:

```
# Do more operations on y
z = y * y * 3
print(z)
z = z.mean()
print(z)

tensor([14.74698, 27.29934, 5.63190], grad_fn=<MulBackward0>)
tensor(15.89274, grad_fn=<MeanBackward0>)
```

In [93]:

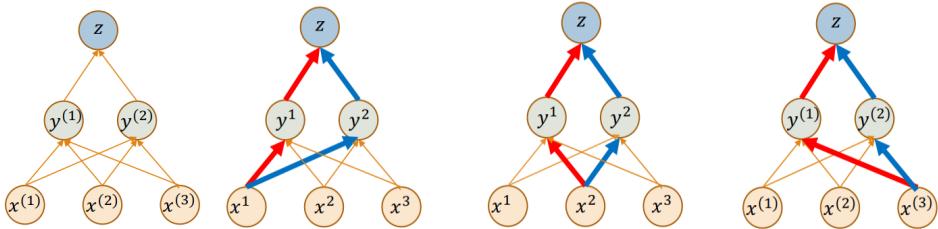
```
# Let's compute the gradients with backpropagation
# When we finish our computation we can call .backward() and have all the gradients computed
# The gradient for this tensor will be accumulated into .grad attribute.
# It is the partial derivate of the function w.r.t. the tensor

z.backward()
print(x.grad) # dz/dx

# Generally speaking, torch.autograd is an engine for computing vector-Jacobian product
# It computes partial derivates while applying the chain rule
```

tensor([4.43426, 6.03317, 2.74029])

Примеры расчета градиента суперпозиции двух функций нескольких переменных:



$$\frac{dz}{dx^1} = \frac{dz}{dy^1} \frac{dy^1}{dx^1} + \frac{dz}{dy^2} \frac{dy^2}{dx^1} \quad \frac{dz}{dx^2} = \frac{dz}{dy^1} \frac{dy^1}{dx^2} + \frac{dz}{dy^2} \frac{dy^2}{dx^2} \quad \frac{dz}{dx^3} = \frac{dz}{dy^1} \frac{dy^1}{dx^3} + \frac{dz}{dy^2} \frac{dy^2}{dx^3}$$

Т.е. нам нужны градиенты по всем возможным путям (рассмотренным в обратном порядке) зависимостей переменных.

Запись этой же задачи в векторной нотации:

$$\bullet \frac{dz}{d\mathbf{x}} = \nabla_{\mathbf{x}}(z) = \begin{pmatrix} \frac{\partial z}{\partial x_1} \\ \dots \\ \frac{\partial z}{\partial x_n} \end{pmatrix} = \left(\frac{dy}{d\mathbf{x}} \right)^T \cdot \nabla_y(z) = J(\mathbf{y}(\mathbf{x}))^T \cdot \nabla_y(z) = J(\mathbf{y}(\mathbf{x}))^T \cdot \begin{pmatrix} \frac{\partial z}{\partial y_1} \\ \dots \\ \frac{\partial z}{\partial y_m} \end{pmatrix}$$

• Где J это Якобиан:

$$J(\mathbf{y}(\mathbf{x})) = \begin{pmatrix} \frac{\partial y_1}{\partial x_1} & \dots & \frac{\partial y_1}{\partial x_n} \\ \vdots & \ddots & \vdots \\ \frac{\partial y_m}{\partial x_1} & \dots & \frac{\partial y_m}{\partial x_n} \end{pmatrix}$$

In [94]:

```
# Model with non-scalar output:  
# If a Tensor is non-scalar (more than 1 elements), we need to specify arguments for backwa  
# specify a gradient argument that is a tensor of matching shape.  
# needed for vector-Jacobian product  
  
x = torch.randn(3, requires_grad=True)  
  
y = x * 2  
for _ in range(10):  
    y = y * 2  
  
print(y)  
print(y.shape)
```

tensor([1292.58887, -2687.01343, 244.47513], grad_fn=<MulBackward0>)
torch.Size([3])

In [95]:

```
v = torch.tensor([0.1, 1.0, 0.0001], dtype=torch.float32)  
y.backward(v)  
print(x.grad)
```

tensor([2.04800e+02, 2.04800e+03, 2.04800e-01])

Stop a tensor from tracking history: For example during our training loop when we want to update our weights then this update operation should not be part of the gradient computation

- `x.requires_grad_(False)`
- `x.detach()`
- wrap in with `torch.no_grad()`:

In [96]:

```
# .requires_grad_(...) changes an existing flag in-place.  
  
a = torch.randn(2, 2)  
print(f'a.requires_grad = {a.requires_grad}')  
  
b = ((a * 3) / (a - 1))  
print(f'b.grad_fn = {b.grad_fn}')  
  
a.requires_grad_(True)  
print(f'a.requires_grad = {a.requires_grad}')  
  
b = (a * a).sum()  
print(f'b.grad_fn = {b.grad_fn}')  
  
a.requires_grad = False  
b.grad_fn = None  
a.requires_grad = True  
b.grad_fn = <SumBackward0 object at 0x00000238229CC988>
```

In [97]:

```
# .detach(): get a new Tensor with the same content but no gradient computation:  
a = torch.randn(2, 2, requires_grad=True)  
print(a.requires_grad)  
  
b = a.detach()  
print(b.requires_grad)
```

True
False

In [98]:

```
# wrap in 'with torch.no_grad():'  
a = torch.randn(2, 2, requires_grad=True)  
print(a.requires_grad)  
  
with torch.no_grad():  
    print((x ** 2).requires_grad)
```

True
False

In []:

```
# backward() accumulates the gradient for this tensor into .grad attribute.  
# !!! We need to be careful during optimization !!!  
# Use .zero_() to empty the gradients before a new optimization step!  
  
weights = torch.ones(4, requires_grad=True)  
  
for epoch in range(3):  
    # just a dummy example  
    # 'forward pass'  
    model_output = (weights*3).sum()  
  
    model_output.backward()  
  
    print(weights.grad)  
  
    # optimize model, i.e. adjust weights...  
    with torch.no_grad():  
        weights -= 0.1 * weights.grad  
  
    # this is important! It affects the final weights & output  
    weights.grad.zero_()  
  
print(weights)  
print(model_output)  
  
# Optimizer has zero_grad() method  
# optimizer = torch.optim.SGD([weights], lr=0.1)  
# During training:  
# optimizer.step()  
# optimizer.zero_grad()
```

Автоматическое выполнение обратного прохода с помощью `l.backward()` :

In [99]:

```
# Модель линейной регрессии (с несколькими параметрами)
#  $f = X * w$ 

# Данные для обучения:
# признаки X: рассматривается 4 наблюдения (ось 0) и 2 признака (ось 1):

X = torch.tensor([[1., 40.],
                  [2., 30.],
                  [3., 20.],
                  [4., 10.]], dtype=torch.float32) # Size([4, 2])

# истинное значение весов (используется только для получения обучающих правильных ответов):
w_ans = torch.tensor([2., 1.], dtype=torch.float32)
# Y - правильные ответы:
Y = X @ w_ans

torch.set_printoptions(precision=5) # точность вывода на печать значений тензоров
print(f'w true value = {w_ans}, Y = {Y}'')

# model (модель, в нашем случае: линейная регрессия)

# изначальное значение весов w
#!!! requires_grad=True
w = torch.tensor([0.0, 0.0], dtype=torch.float32, requires_grad=True)

# прямое распространение:
def forward(X):
    return X @ w # Size([4])

# Loss = MSE (функция потерь, в нашем случае: средняя квадратичная ошибка)
def loss(y, y_pred):
    return ((y_pred - y)**2).mean() # Size([])

# Training
learning_rate = 0.0013
n_iters = 1000 + 1

# основной цикл:
for epoch in range(n_iters):
    # predict = forward pass
    y_pred = forward(X)

    # loss
    l = loss(Y, y_pred)

    #!!! backward pass
    # calculate gradients = backward pass
    l.backward()

    # update weights
    #w.data = w.data - learning_rate * w.grad
    with torch.no_grad():
        w -= learning_rate * w.grad

    # zero the gradients after updating
    w.grad.zero_()
```

```
if epoch % 5 == 0:  
    print(f'epoch {epoch}: w = {w}, y_pred = {y_pred}, loss = {l:.8f}\ngradient = {dw}'  
  
w true value = tensor([2., 1.]), Y = tensor([42., 34., 26., 18.])  
epoch 0: w = tensor([0.16900, 2.21000], requires_grad=True), y_pred = tensor([0., 0., 0., 0.], grad_fn=<MvBackward>), loss = 980.00000000  
gradient = tensor([-0.00029, 0.00038])  
epoch 5: w = tensor([0.13813, 0.24422], requires_grad=True), y_pred = tensor([81.70274, 61.57523, 41.44772, 21.32021], grad_fn=<MvBackward>), loss = 646.58898926  
gradient = tensor([-0.00029, 0.00038])  
epoch 10: w = tensor([0.33966, 1.82453], requires_grad=True), y_pred = tensor([15.22920, 11.70164, 8.17407, 4.64651], grad_fn=<MvBackward>), loss = 427.49307251  
gradient = tensor([-0.00029, 0.00038])  
epoch 15: w = tensor([0.34364, 0.53338], requires_grad=True), y_pred = tensor([68.78386, 52.09385, 35.40385, 18.71384], grad_fn=<MvBackward>), loss = 283.42614746  
gradient = tensor([-0.00029, 0.00038])  
epoch 20: w = tensor([0.49880, 1.56850], requires_grad=True), y_pred = tensor([25.13912, 19.37721, 13.61531, 7.85340], grad_fn=<MvBackward>), loss = 188.61228943
```

Использование встроенного оптимизатора `optimizer = torch.optim.SGD([w], lr=learning_rate)` и функции потерь `loss = nn.MSELoss()`:

In [100]:

```
import torch
import torch.nn as nn

# Модель линейной регрессии (с несколькими параметрами)
#  $f = X * w$ 

#-----
# 0) Training samples

# Данные для обучения:
# признаки X: рассматривается 4 наблюдения (ось 0) и 2 признака (ось 1):
X = torch.tensor([[1., 40.],
                  [2., 30.],
                  [3., 20.],
                  [4., 10.]], dtype=torch.float32) # Size([4, 2])

# истинное значение весов (используется только для получения обучающих правильных ответов):
w_ans = torch.tensor([2., 1.], dtype=torch.float32)
# Y - правильные ответы:
Y = X @ w_ans

torch.set_printoptions(precision=5) # точность вывода на печать значений тензоров
print(f'w true value = {w_ans}, Y = {Y}')
```

#-----

```
# 1) Design Model: Weights to optimize and forward function

# изначальное значение весов w
w = torch.tensor([0.0, 0.0], dtype=torch.float32, requires_grad=True)

# model (модель, в нашем случае: линейная регрессия)
# прямое распространение:
def forward(X):
    return X @ w # Size([4])

#-----
# 2) Define Loss and optimizer

# callable function
loss = nn.MSELoss()

# Loss = MSE (функция потерь, в нашем случае: средняя квадратичная ошибка)
# def loss(y, y_pred):
#     return ((y_pred - y)**2).mean() # Size([])

learning_rate = 0.0013
optimizer = torch.optim.SGD([w], lr=learning_rate)

#-----
# 3) Training Loop
# основной цикл:
n_iters = 1000 + 1

for epoch in range(n_iters):
    # predict = forward pass
    y_pred = forward(X)
```

```

# Loss
l = loss(Y, y_pred)

# calculate gradients = backward pass
l.backward()

# update weights
optimizer.step()

# zero the gradients after updating
optimizer.zero_grad()

if epoch % 5 == 0:
    print(f'epoch {epoch}: w = {w}, y_pred = {y_pred}, loss = {l:.8f}\ngradient = {dw}')

```

```

w true value = tensor([2., 1.]), Y = tensor([42., 34., 26., 18.])
epoch 0: w = tensor([0.16900, 2.21000], requires_grad=True), y_pred = tensor([0., 0., 0., 0.], grad_fn=<MvBackward>), loss = 980.00000000
gradient = tensor([-0.00029, 0.00038])
epoch 5: w = tensor([0.13813, 0.24422], requires_grad=True), y_pred = tensor([81.70274, 61.57523, 41.44772, 21.32021], grad_fn=<MvBackward>), loss =
646.58898926
gradient = tensor([-0.00029, 0.00038])
epoch 10: w = tensor([0.33966, 1.82453], requires_grad=True), y_pred = tensor([15.22920, 11.70164, 8.17407, 4.64651], grad_fn=<MvBackward>), loss =
427.49307251
gradient = tensor([-0.00029, 0.00038])
epoch 15: w = tensor([0.34364, 0.53338], requires_grad=True), y_pred = tensor([68.78386, 52.09385, 35.40385, 18.71384], grad_fn=<MvBackward>), loss =
283.42614746
gradient = tensor([-0.00029, 0.00038])
epoch 20: w = tensor([0.49880, 1.56850], requires_grad=True), y_pred = tensor([25.13912, 19.37721, 13.61531, 7.85340], grad_fn=<MvBackward>), loss =
188.61228943
...

```

Использование модели:

`torch.nn.Linear(in_features, out_features, bias=True)` Applies a linear transformation to the incoming data: $y = xA^T + b$

Parameters:

- `in_features` – size of each input sample
- `out_features` – size of each output sample
- `bias` – If set to False, the layer will not learn an additive bias. Default: True

In [101]:

```
# X_test = torch.tensor([[1], [2], [3], [4]], dtype=torch.float32) # torch.tensor([5], dtype=torch.float32)

X = torch.tensor([[1., 40.],
                 [2., 30.],
                 [3., 20.],
                 [4., 10.]], dtype=torch.float32) # Size([4, 2])

n_samples, n_features = X.shape

print(n_samples, n_features)

# n_samples, n_features = X_test.shape
# input_size = n_features
# output_size = n_features

class LinearRegression(nn.Module):
    def __init__(self, input_dim, output_dim):
        super(LinearRegression, self).__init__()
        # define different layers
        self.lin = nn.Linear(input_dim, output_dim, bias=False)
    def forward(self, x):
        return self.lin(x)

model = LinearRegression(n_features, 1)

print(f'Prediction before training: f(5) = {model(X)}')
```

4 2

Prediction before training: f(5) = tensor([[6.26964],
[4.58868],
[2.90772],
[1.22675]], grad_fn=<MmBackward>)

In [102]:

```
X_test.shape
```

```
-----  
NameError                                 Traceback (most recent call last)  
<ipython-input-102-cf290153e199> in <module>  
----> 1 X_test.shape
```

NameError: name 'X_test' is not defined

In [103]:

```
import torch
import torch.nn as nn

# Модель линейной регрессии (с несколькими параметрами)
#  $f = X * w$ 

#-----
# 0) Training samples

# Данные для обучения:
# признаки X: рассматривается 4 наблюдения (ось 0) и 2 признака (ось 1):
X = torch.tensor([[1., 40.],
                  [2., 30.],
                  [3., 20.],
                  [4., 10.]], dtype=torch.float32) # Size([4, 2])

print(f'X.shape = {X.shape}')
X_samples, X_features = X.shape

# истинное значение весов (используется только для получения обучающих правильных ответов):
w_ans = torch.tensor([2., 1.], dtype=torch.float32)
# Y - правильные ответы:
Y = X @ w_ans
print(f'Y.shape = {Y.shape}')
Y_features = 1

torch.set_printoptions(precision=5) # точность вывода на печать значений тензоров
print(f'w true value = {w_ans}, Y = {Y}')

#-----
# 1) Design Model, the model has to implement the forward pass!
# Here we can use a built-in model from PyTorch
input_size = n_features
output_size = n_features

# we can call this model with samples X
# model = nn.Linear(input_size, output_size)

class LinearRegression(nn.Module):
    def __init__(self, input_dim, output_dim):
        super(LinearRegression, self).__init__()
        # define different layers
        self.lin = nn.Linear(input_dim, output_dim, bias=False)
    def forward(self, x):
        return self.lin(x)

model = LinearRegression(X_features, Y_features)

print(f'Prediction before training: f({X}) = {model(X)}')

# # model (модель, в нашем случае: линейная регрессия)
# # прямое распространение:
# def forward(X):
#     return X @ w # Size([4])

#-----
# 2) Define Loss and optimizer
```

```

# callable function
criterion = nn.MSELoss()

learning_rate = 0.0013
optimizer = torch.optim.SGD([w], lr=learning_rate)

#-----
# 3) Training Loop
# основной цикл:
n_iters = 1000 + 1

for epoch in range(n_iters):
    # Forward pass and Loss
    y_predicted = model(X)
    loss = criterion(y_predicted, Y)

    # Backward pass and update
    loss.backward()
    optimizer.step()

    # zero grad before new step
    optimizer.zero_grad()

    if epoch % 5 == 0:
        print(f'epoch {epoch}: w = {w}, y_pred = {y_pred}, loss = {loss:.8f}\ngradient = {dw}'

```

```

X.shape = torch.Size([4, 2])
Y.shape = torch.Size([4])
w true value = tensor([2., 1.]), Y = tensor([42., 34., 26., 18.])
Prediction before training: f(tensor([[ 1., 40.],
[ 2., 30.],
[ 3., 20.],
[ 4., 10.]])) = tensor([-24.89012],
[-18.63989],
[-12.38966],
[-6.13942]), grad_fn=<MmBackward>
epoch 0: w = tensor([1.99996, 1.00000], requires_grad=True), y_pred = tens
or([42.00007, 34.00001, 25.99994, 17.99988], grad_fn=<MvBackward>), loss =
0.00000001
gradient = tensor([-0.00029, 0.00038])
epoch 5: w = tensor([1.99996, 1.00000], requires_grad=True), y_pred = tens
or([42.00007, 34.00001, 25.99994, 17.99988], grad_fn=<MvBackward>), loss =
0.00000001
gradient = tensor([-0.00029, 0.00038])
epoch 10: w = tensor([1.99996, 1.00000], requires_grad=True), y_pred = ten

```

Спасибо за внимание!

Технический раздел:

In []:

In []:

- И Введение в искусственные нейронные сети
 - Базовые понятия и история
- И Машинное обучение и концепция глубокого обучения
- И Почему глубокое обучение начало приносить плоды и активно использоваться только после 2010 г?
 - Производительность оборудования
 - Доступность наборов данных и тестов
 - Алгоритмические достижения в области глубокого обучения
 - Улучшенные подходы к регуляризации
 - Улучшенные схемы инициализации весов
 - (повтор) Усовершенствованные методы градиентного спуска
- Обратное распространение ошибки
 - Оптимизация
 - Стохастический градиентный спуск
 - Усовершенствованные методы градиентного спуска
- Введение в PyTorch

next **Q:** qs line

next **A:** an line

next **Note:** an line

next **Def:** df line

next **Ex:** ex line

next **+ pl** line

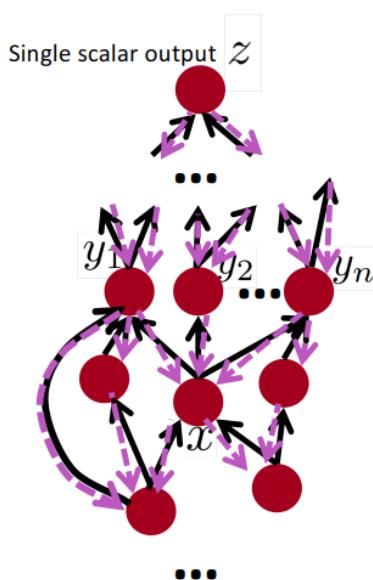
next **- mn** line

next **± plmn** line

next **⇒ hn** line

- Работа с графом потока вычислений нужна для того, чтобы решить **задачу обучения многослойной ИНС**. А эта задача требует после получения результатов и оценки ошибки **выполнения обратного прохода** дающего градиент ошибки для весов (параметров) модели и последующей процедуры оптимизации весов.

Back-Prop in General Computation Graph



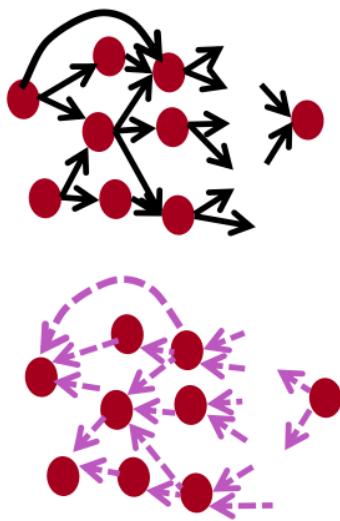
1. Fprop: visit nodes in topological sort order
 - Compute value of node given predecessors
2. Bprop:
 - initialize output gradient = 1
 - visit nodes in reverse order:
 - Compute gradient wrt each node using gradient wrt successors
 - $\{y_1, y_2, \dots, y_n\}$ = successors of x

$$\frac{\partial z}{\partial x} = \sum_{i=1}^n \frac{\partial z}{\partial y_i} \frac{\partial y_i}{\partial x}$$

Done correctly, big O() complexity of fprop and bprop is **the same**

In general our nets have regular layer-structure and so we can use matrices and Jacobians...

Automatic Differentiation

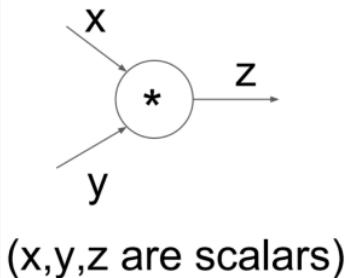


- The gradient computation can be automatically inferred from the symbolic expression of the fprop
- Each node type needs to know how to compute its output and how to compute the gradient wrt its inputs given the gradient wrt its output
- Modern DL frameworks (Tensorflow, PyTorch, etc.) do backpropagation for you but mainly leave layer/node writer to hand-calculate the local derivative

Backprop Implementations

```
class ComputationalGraph(object):
    ...
    def forward(inputs):
        # 1. [pass inputs to input gates...]
        # 2. forward the computational graph:
        for gate in self.graph.nodes_topologically_sorted():
            gate.forward()
        return loss # the final gate in the graph outputs the loss
    def backward():
        for gate in reversed(self.graph.nodes_topologically_sorted()):
            gate.backward() # little piece of backprop (chain rule applied)
        return inputs_gradients
```

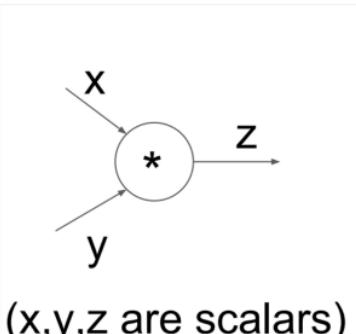
Implementation: forward/backward API



```
class MultiplyGate(object):
    def forward(x,y):
        z = x*y
        return z
    def backward(dz):
        # dx = ... #todo
        # dy = ... #todo
        return [dx, dy]
```

$$\frac{\partial L}{\partial z}$$
$$\frac{\partial L}{\partial x}$$

Implementation: forward/backward API



```
class MultiplyGate(object):
    def forward(x,y):
        z = x*y
        self.x = x # must keep these around!
        self.y = y
        return z
    def backward(dz):
        dx = self.y * dz # [dz/dx * dL/dz]
        dy = self.x * dz # [dz/dy * dL/dz]
        return [dx, dy]
```

Gradient checking: Numeric Gradient

- For small h ($\approx 1e-4$), $f'(x) \approx \frac{f(x + h) - f(x - h)}{2h}$
- Easy to implement correctly
- But approximate and **very slow**:
 - Have to recompute f for **every parameter** of our model
- Useful for checking your implementation
 - In the old days when we hand-wrote everything, it was key to do this everywhere.
 - Now much less needed, when throwing together layers

In []:

|