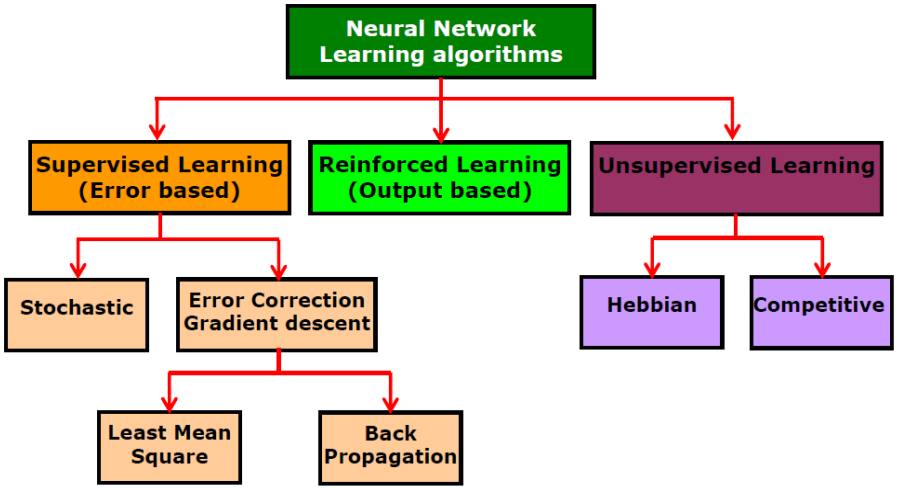
Нейронные сети и их практическое применение.

Лекция 4. Алгоритм обратного распространения ошибки.

Дмитрий Буряк
к.ф.-м.н
dyb04@yandex.ru

Классификация алгоритмов обучения

1. Обучение с учителем (Supervised learning). Обучающая выборка содержит входные сигналы, и ожидаемые выходы. Цель подбора весов – близость реальных и ожидаемых выходов сети.
2. Обучение без учителя (Unsupervised learning). Нет знаний о желаемых выходах сети.
   - конкуренция нейронов между собой:
     Winner takes all (WTA);
     Winner takes most (WTM).
   - учет корреляции обучающих и выходных сигналов (обучение по Хеббу).
3. Reinforcement learning. Имеется только информация о правильности выхода сети



Алгоритм обратного распространения ошибки (1)

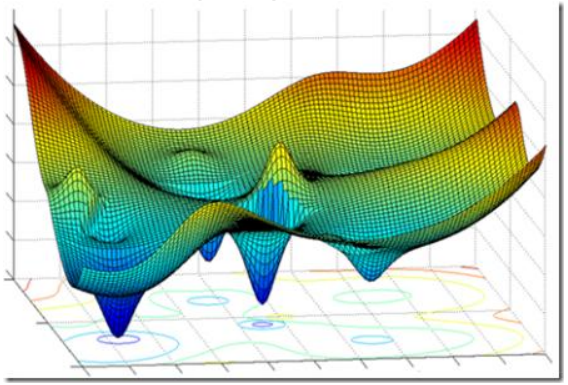
- Функция ошибки $E = F(d, y(x, w))$

  x – известные входные вектора;

  d – требуемые выходные значения сети;

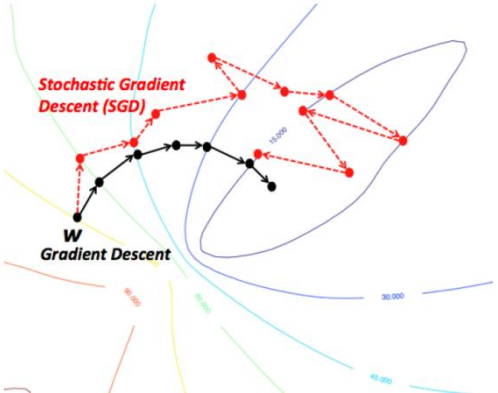
  $y(x,w)$ – вычисленные выходные значения сети;

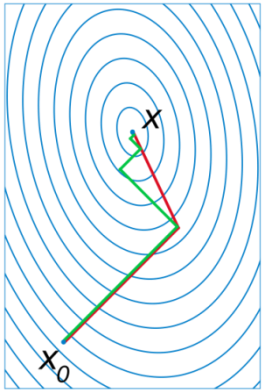
  w – вектор весов сети.



- Поиск минимума функции ошибки

$$\hat{w} = \arg\min_{w} F(d, y(x, w))$$



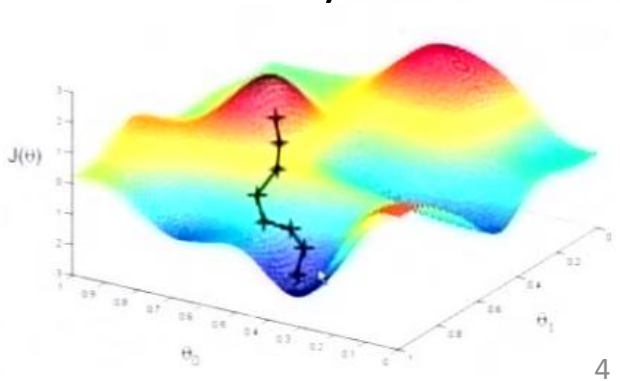


Алгоритм обратного распространения ошибки (2)

1. Инициализировать веса сети
2. Выбрать обучающую пару из обучающего множества.
3. Подать на вход сети, вычислить выход
4. Вычислить разность между реальным выходом и требуемым выходом
5. Подкорректировать веса сети, так чтобы уменьшить ошибку
6. Повторять шаги 2-5 для каждого вектора из обучающего множества, пока ошибка не достигнет приемлемого уровня

Шаги 2, 3 – проход вперед

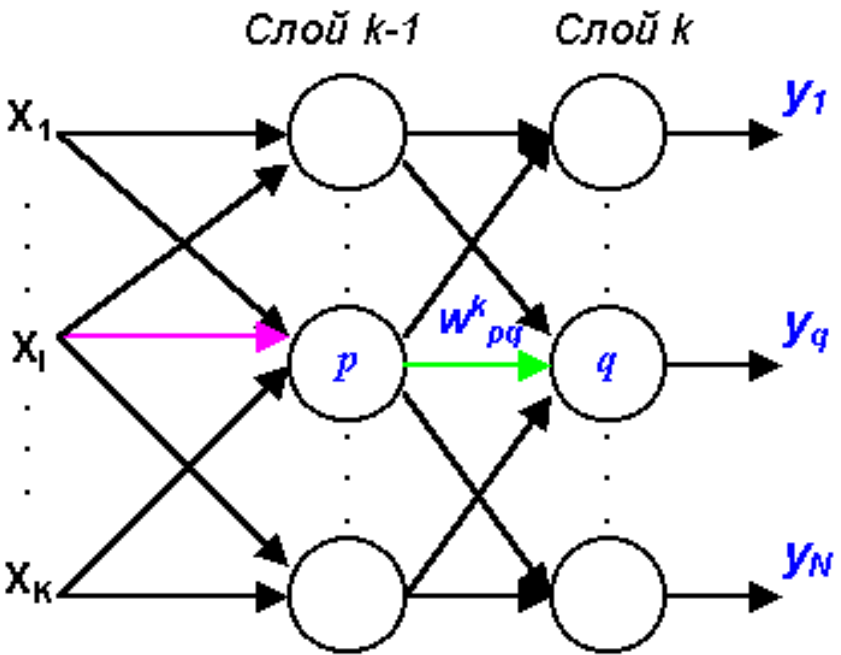
Шаги 4, 5 – обратный проход



Алгоритм обратного распространения ошибки (3)

Необходимое условие: дифференцируемость функции активации.

$$E(w) = \frac{1}{2}\sum_j (y_j - d_j)^2$$



Выходной слой

$$\delta_q = \frac{dy_q}{du_q}(y_q - d_q)$$

Скрытый слой

$$\delta_p^{k-1} = \left[\sum_q \delta_q^k \cdot w_{pq}^k\right] \cdot \frac{d\nu_p^{(k-1)}}{du_p^{(k-1)}}$$

$$\Delta w_{pq}^k = \eta \delta_q^k \nu_p^{k-1}$$

$$w_{pq}^k(t+1) = w_{pq}^k(t) + \Delta w_{pq}^k$$

Алгоритм обратного распространения ошибки (4)

Минимизация целевой функции : $E(w) = \frac{1}{2}\sum_j (y_j - d_j)^2$

d - желаемый выход

y - реальный выход

Метод обучения - градиентный спуск.

$$\Delta w_{lp}^{(k-1)} = -\eta \cdot \frac{\partial E}{\partial w_{lp}^{(k-1)}} \qquad \frac{\partial E}{\partial w_{lp}^{(k-1)}} = \frac{\partial E}{\partial v_p^{(k-1)}} \cdot \frac{dv_p^{(k-1)}}{du_p^{(k-1)}} \cdot \frac{\partial u_p^{(k-1)}}{\partial w_{lp}^{(k-1)}}$$

$$\frac{\partial E}{\partial v_p^{(k-1)}} = \sum_q \frac{\partial E}{\partial v_q^{(k)}} \cdot \frac{dv_q^{(k)}}{du_q^{(k)}} \cdot \frac{\partial u_q^{(k)}}{\partial v_p^{(k-1)}} = \sum_q \frac{\partial E}{\partial v_q^{(k)}} \cdot \frac{dv_q^{(k)}}{du_q^{(k)}} \cdot w_{pq}^{(k)} \qquad \delta_q^k = \frac{\partial E}{\partial v_q^{(k)}} \cdot \frac{dv_q^{(k)}}{du_q^{(k)}}$$

$$\delta_p^{k-1} = \left[\sum_q \delta_q^k \cdot w_{pq}^k\right] \cdot \frac{dv_p^{(k-1)}}{du_p^{(k-1)}}$$

Выходной слой

$$\delta_i = (y_i - d_i) \cdot \frac{dy_i}{du_i}$$

$$\Delta w_{pq}^k = -\eta \delta_q^k v_p^{k-1}$$

Алгоритм обратного распространения ошибки (5)

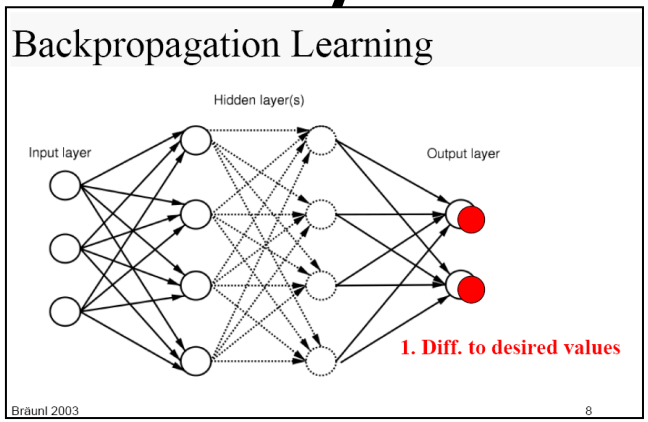
1. Рассчитать выходы сети.
2. Рассчитать для выходного слоя:
3. Рассчитать для всех остальных слоев:
4. Скорректировать веса:
5. Если ошибка существенна, перейти на шаг 1.

$$\delta_i = (y_i - d_i) \cdot \frac{dy_i}{du_i} \qquad \Delta w_{pi}^N = -\eta \delta_i \nu_p^{N-1}$$

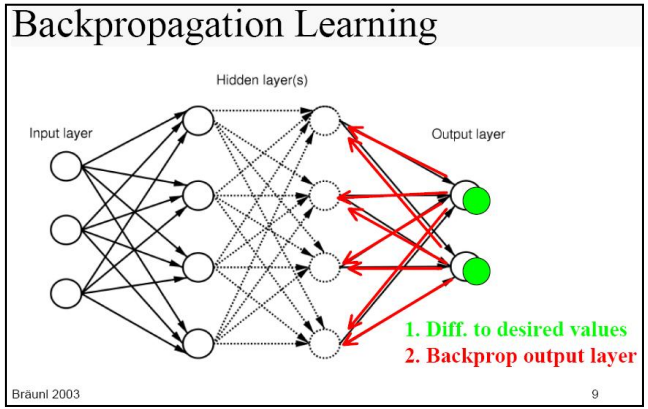
$$\delta_p^{k-1} = \left[\sum_q \delta_q^k \cdot w_{pq}^k \right] \cdot \frac{d\nu_p^{(k-1)}}{du_p^{(k-1)}} \qquad \Delta w_{pq}^k = -\eta \delta_q^k \nu_p^{k-1}$$

$$w_{pq}^k(t+1) = w_{pq}^k(t) + \Delta w_{pq}^k$$

Визуализация. Выходной слой



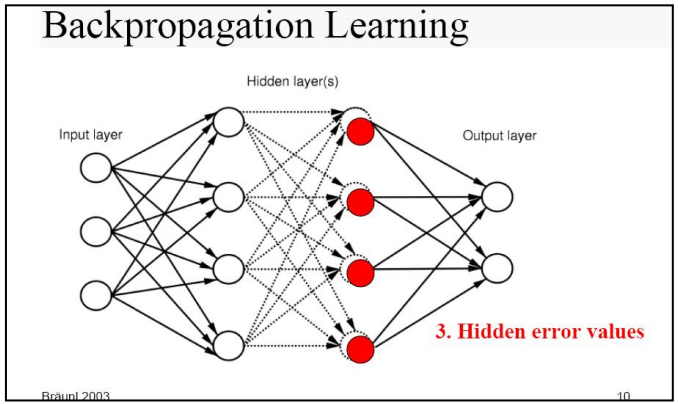
$$\delta_i = (y_i - d_i) \cdot \frac{dy_i}{du_i}$$



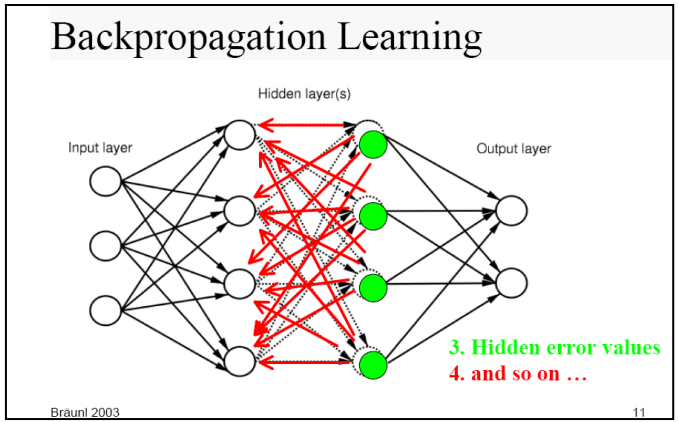
$$\delta_p^{N-1} = \left[\sum_i \delta_i \cdot w_{pi}^N \right] \cdot \frac{d\nu_p^{(N-1)}}{du_p^{(N-1)}}$$

$$\Delta w_{pi}^N = -\eta \delta_i \, \nu_p^{N-1}$$

Визуализация. Внутренний слой



$$\delta_p^{k-1} = \left[\sum_q \delta_q^k \cdot w_{pq}^k \right] \cdot \frac{d\nu_p^{(k-1)}}{du_p^{(k-1)}}$$

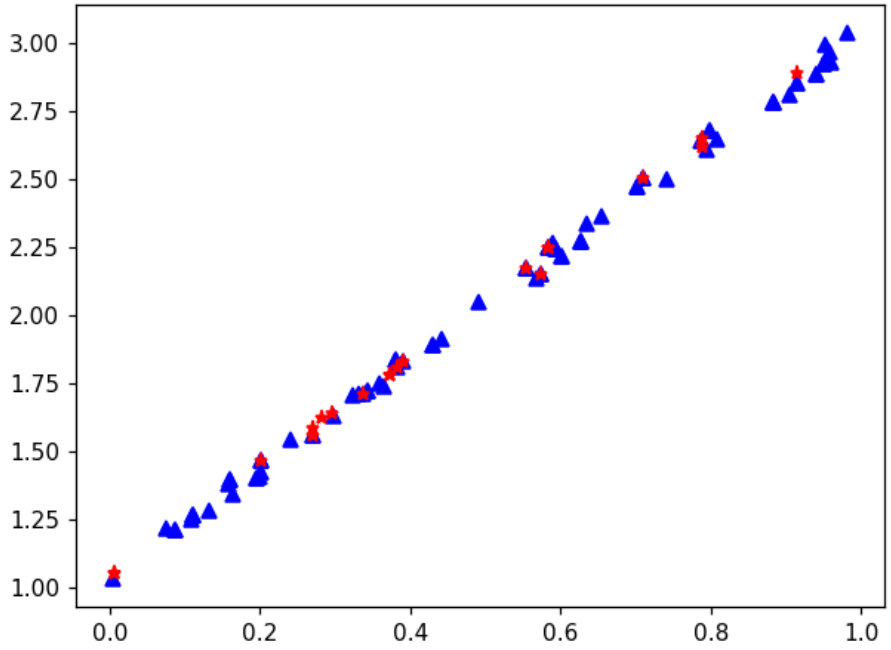


$$\delta_p^{k-2} = \left[\sum_q \delta_q^{k-1} \cdot w_{pq}^{k-1} \right] \cdot \frac{d\nu_p^{(k-2)}}{du_p^{(k-2)}}$$

$$\Delta w_{pq}^{k-1} = -\eta \delta_q^{k-1} \nu_p^{k-2}$$

Пример. Линейная регрессия (1)

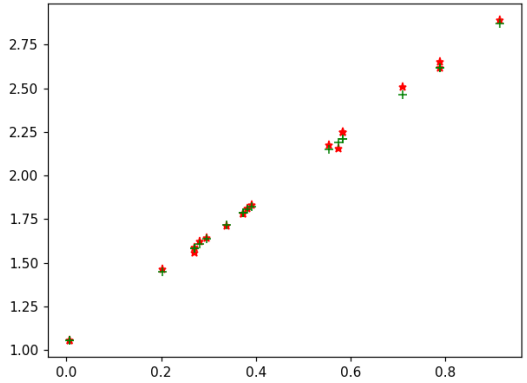
```
import numpy
import torch
import torch.optim as optim
from matplotlib import pyplot as plt

# Data Generation
torch.manual_seed(42)
x = torch.rand(100, 1)
y = 1 + 2 * x + 0.1 * torch.rand(100, 1)

# Shuffles the indices
idx = (torch.rand(100)*100).long()

# Train-valid split
train_idx = idx[:80]
val_idx = idx[80:]
x_train, y_train = x[train_idx], y[train_idx]
x_val, y_val = x[val_idx], y[val_idx]

plt.figure("train")
plt.plot(x_train.numpy(),y_train.numpy(),"b",lw=0,marker="^")
plt.plot(x_val.numpy(),y_val.numpy(),"r",lw=0,marker="*")
plt.show()
```



Пример. Линейная регрессия (2)

```
torch.manual_seed(42)
a = torch.randn(1, requires_grad=True, dtype=torch.float)
b = torch.randn(1, requires_grad=True, dtype=torch.float)
print("Initial values:",a, b)
lr = 1e-1
n_epochs = 1000
# Defines a SGD optimizer to update the parameters
optimizer = optim.SGD([a, b], lr=lr)

for epoch in range(n_epochs):
    yhat = a + b * x_train
    error = y_train - yhat
    loss = (error ** 2).mean()

    loss.backward()

    #     a -= lr * a.grad
    #     b -= lr * b.grad
    optimizer.step()
    optimizer.zero_grad()
```

```
print("optimized values:",a,b)
with torch.no_grad():
    yhat = a + b * x_train
    yres = a + b * x_val
print("train error:",torch.mean((y_train - yhat)**2))
print("validation error:",torch.mean((y_val- yres)**2))
plt.figure("result")
plt.plot(x_val.numpy(),y_val.numpy(),"r",lw=0,marker="*")
plt.plot(x_val.numpy(),yres.numpy(),"g",lw=0,marker="+")
plt.show()
```

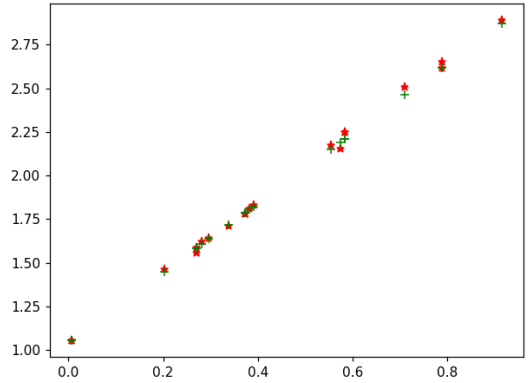



```
Initial values: tensor([0.3367], requires_grad=True) tensor([0.1288], requires_grad=True)
optimized values: tensor([1.0422], requires_grad=True) tensor([1.9991], requires_grad=True)
train error: tensor(0.0007)
validation error: tensor(0.0005)
```

Пример. Линейная регрессия (2)

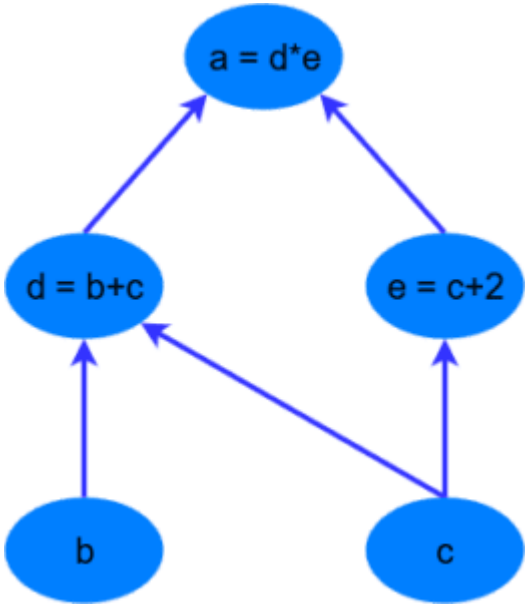
```
torch.manual_seed(42)
a = torch.randn(1, requires_grad=True, dtype=torch.float)
b = torch.randn(1, requires_grad=True, dtype=torch.float)
print("Initial values:",a, b)
lr = 1e-1
n_epochs = 1000
# Defines a SGD optimizer to update the parameters
optimizer = optim.SGD([a, b], lr=lr)

for epoch in range(n_epochs):
    yhat = a + b * x_train
    error = y_train - yhat
    loss = (error ** 2).mean()

    loss.backward()

    #     a -= lr * a.grad
    #     b -= lr * b.grad
    optimizer.step()
    optimizer.zero_grad()
```

```
print("optimized values:",a,b)
with torch.no_grad():
    yhat = a + b * x_train
    yres = a + b * x_val
print("train error:",torch.mean((y_train - yhat)**2))
print("validation error:",torch.mean((y_val- yres)**2))
plt.figure("result")
plt.plot(x_val.numpy(),y_val.numpy(),"r",lw=0,marker="*")
plt.plot(x_val.numpy(),yres.numpy(),"g",lw=0,marker="+")
plt.show()
```




```
Initial values: tensor([0.3367], requires_grad=True) tensor([0.1288], requires_grad=True)
optimized values: tensor([1.0422], requires_grad=True) tensor([1.9991], requires_grad=True)
train error: tensor(0.0007)
validation error: tensor(0.0005)
```

Динамический граф вычислений



*a = (b + c) * (c + 2)*

- Выражение разбивается на части, которые вычисляются последовательно.
- Каждый узел графа: независимый кусок кода, которому для работы нужны лишь входы.
- Преимущества: параллельная обработка.
- Технически: тяжелые вычисления (свертки, умножения матриц) реализованы на C/C++ / CUDA C++.

Тензоры

❑ Основная вычислительная единица - тензор, можно представить как вложение произвольного количества динамических массивов переменной размерности.

❑ Инициализация: 3 матрицы размера 2x2

```
In [1]: import torch
In [2]: x=torch.Tensor(3,2,2)
In [3]: print(x)
tensor([[[8.7725e+29, 3.0904e-41],
         [0.0000e+00, 0.0000e+00]],

        [[       nan, 0.0000e+00],
         [1.3733e-14, 6.4069e+02]],

        [[4.3066e+21, 1.1824e+22],
         [4.3066e+21, 6.3828e+28]]])
```

❑ Матричное умножение

```
In [18]: a=torch.rand(2,3)
In [19]: b=torch.rand(3,2)
In [20]: c=torch.mm(b,a)
In [21]: print(c)
Out[21]:
tensor([[0.4033, 0.0286, 0.1627],
        [0.6677, 0.0475, 0.2699],
        [0.5106, 0.0588, 0.3515]])
```

```
In [38]: a=a*0+1
In [39]: b=b*0+2
In [40]: 3+torch.mm(b,a)
Out[40]:
tensor([[7., 7., 7.],
        [7., 7., 7.],
        [7., 7., 7.]])
```

Тензоры и вычислительные устройства

- 2 типа: на CPU и на GPU

```
In [73]: a=torch.rand(3,3)
In [74]: a
Out[74]:
tensor([[0.5106, 0.9224, 0.0468],
        [0.0877, 0.6954, 0.2929],
        [0.0226, 0.1924, 0.0517]])
In [75]: a=a.cuda(device = 0)
In [76]: a
Out[76]:
tensor([[0.5106, 0.9224, 0.0468],
        [0.0877, 0.6954, 0.2929],
        [0.0226, 0.1924, 0.0517]],
device='cuda:0')
In [77]: a=a.cpu()
In [78]: a
Out[78]:
tensor([[0.5106, 0.9224, 0.0468],
        [0.0877, 0.6954, 0.2929],
        [0.0226, 0.1924, 0.0517]])
```

```
In [79]: a=torch.rand(3,3)

In [80]: b=torch.rand(3,3).cuda(0)

In [81]: a+b
---------------------------------------------------------------------------
RuntimeError                              Traceback (most recent call last)
<ipython-input-81-ca730b97bf8a> in <module>
----> 1 a+b
RuntimeError: expected type torch.FloatTensor but got
torch.cuda.FloatTensor
```

- Операции между тензорами должны выполняться на одном устройстве

  torch.cuda.is_available()

  torch.cuda.device_count()

Автоматическое дифференцирование

```
In [42]: from torch.autograd import Variable
In [43]: x=Variable(torch.ones(2,2)*2,requires_grad = True)
In [44]: z = 2 * (x * x) + 5 * x
In [45]: z.backward(torch.ones(2, 2))
In [46]: print(x.grad)
tensor([[13., 13.],
        [13., 13.]])
In [47]: z
Out[47]:
tensor([[18., 18.],
        [18., 18.]], grad_fn=<AddBackward0>)
```

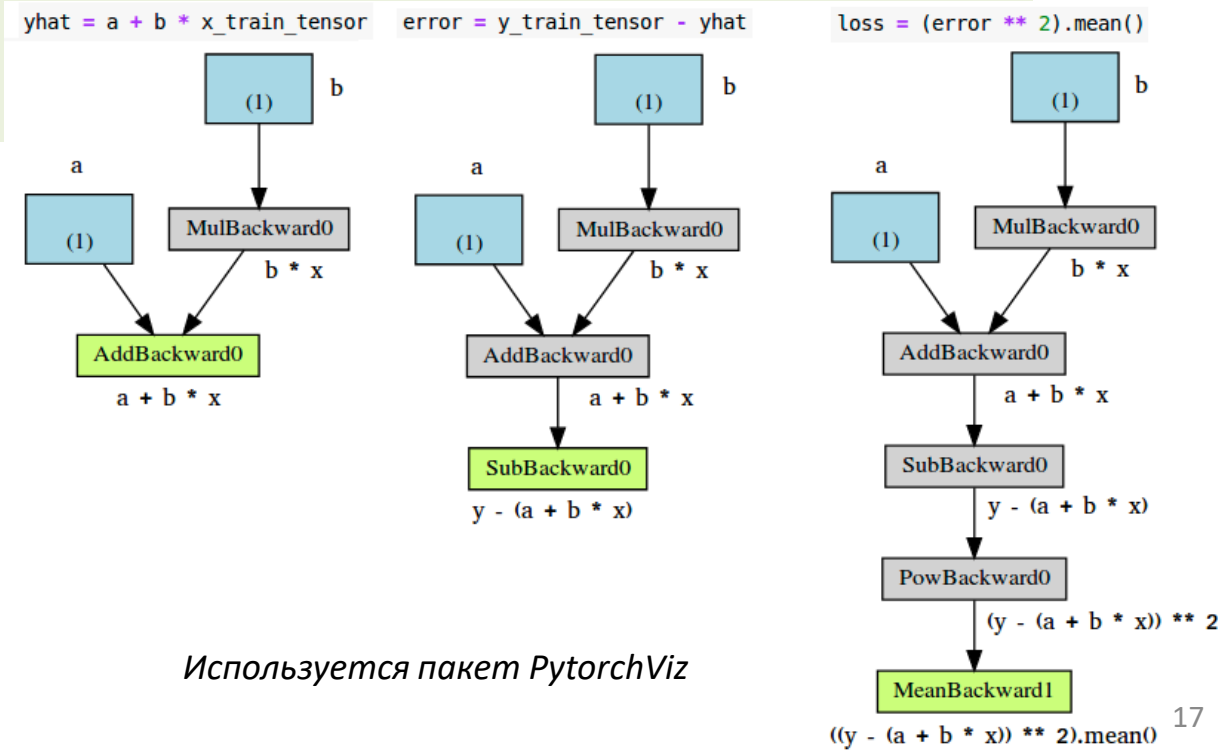
dz/dx = 4x+5,
При x_{ij} =2 dz/dx должен быть заполнен числами 13

Компонент тензора .grad содержит градиент:
значение или функцию

- Как это реализовано: все математические операции (+,-,*,/…) с тензорами перегружены.
- Каждая операция с тензорами проверяет, нужно ли вычислять градиент (компонент класса torch.Tensor requires_grad == True)
- Каждый torch.Tensor содержит компоненты:
  grad (значение градиента) или
  grad_fn(ссылка на функцию вычисления градиента)
- Вызов z.backward() запускает проход по графу вычислений в обратную сторону, с вызовом функций grad_fn

Примеры вычислительных графов в представлении Pytorch (1)

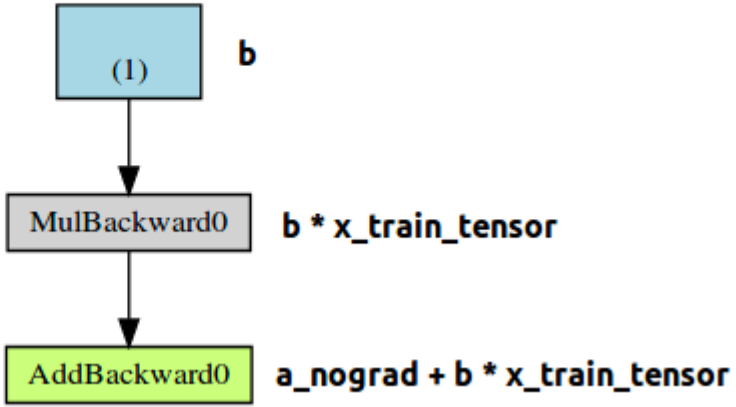
```
a = torch.randn(1, requires_grad=True, dtype=torch.float, device=device)
b = torch.randn(1, requires_grad=True, dtype=torch.float, device=device)
yhat = a + b * x_train_tensor
error = y_train_tensor - yhat
loss = (error ** 2).mean()
```



Используется пакет PytorchViz

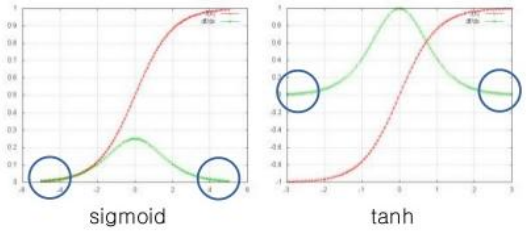
Примеры вычислительных графов в представлении Pytorch (2)

```
a_nograd = torch.randn(1, requires_grad=False, dtype=torch.float, device=device)
b = torch.randn(1, requires_grad=True, dtype=torch.float, device=device)

yhat = a_nograd + b * x_train_tensor
```

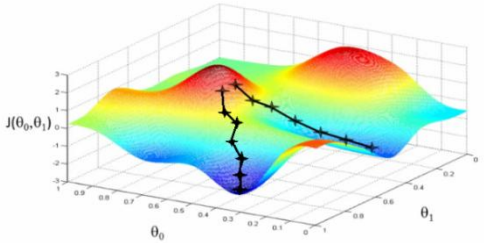


Проблемы алгоритма обратного распространения ошибки

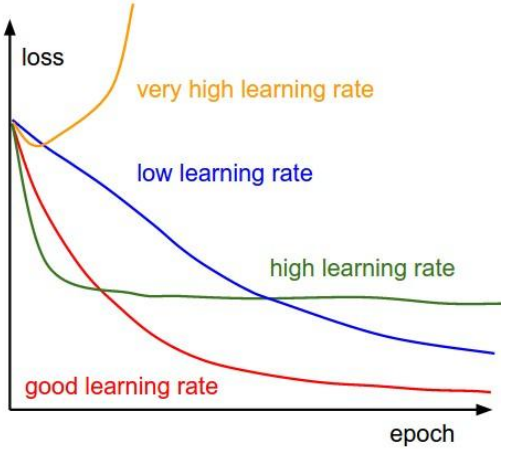
1. Градиент стремится к 0.



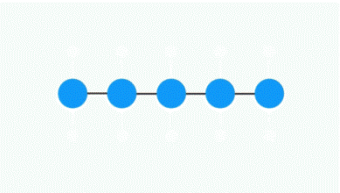
2. Обнаружение локального минимума.



3. Выбор скорости обучения.



4. Исчезающий градиент.



Вопросы

1. Типы алгоритмов обучения?
2. Почему нейроны с сигмоидальными функциями активации могут замедлять процесс обучения?
3. Какие основные проблемы алгоритма обратного распространения ошибки?