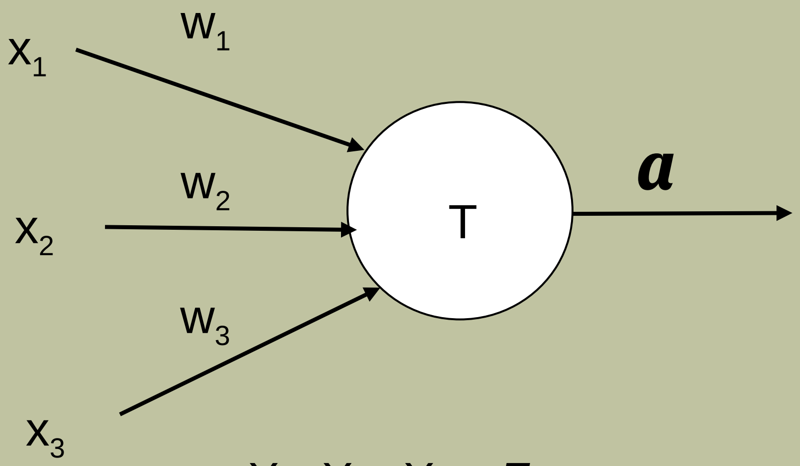
Лекция 8

Использование GPU при *глубоком обучении*.

- **Нейросети и глубокое обучение.**

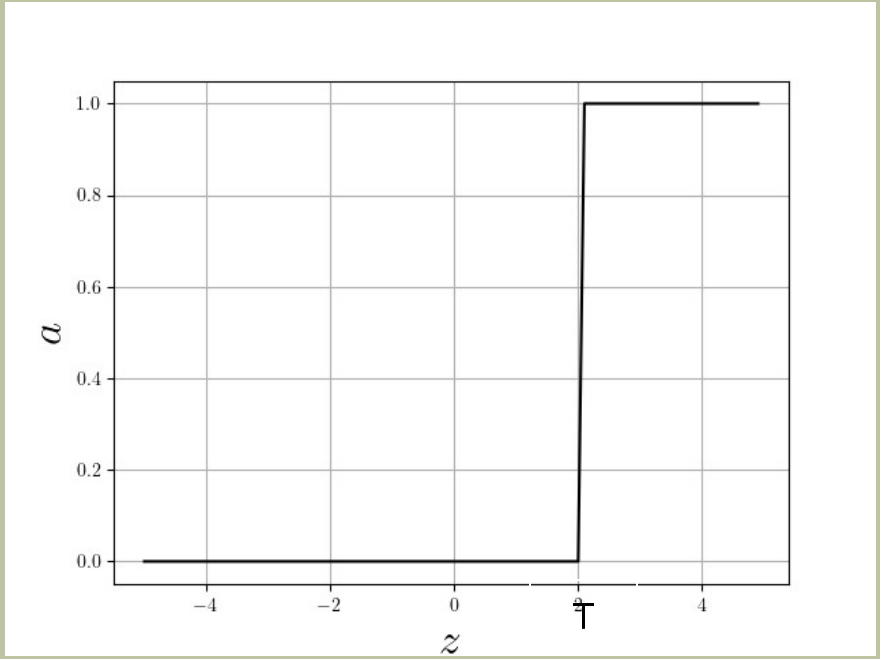
Искусственные нейроны.

Перцептроны и вентили. Логистические нейроны.

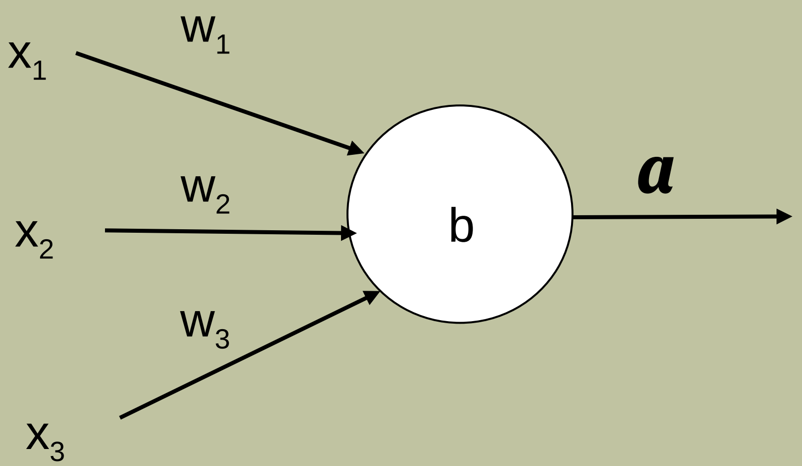


$X_1, X_2, X_3 \in Z$

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

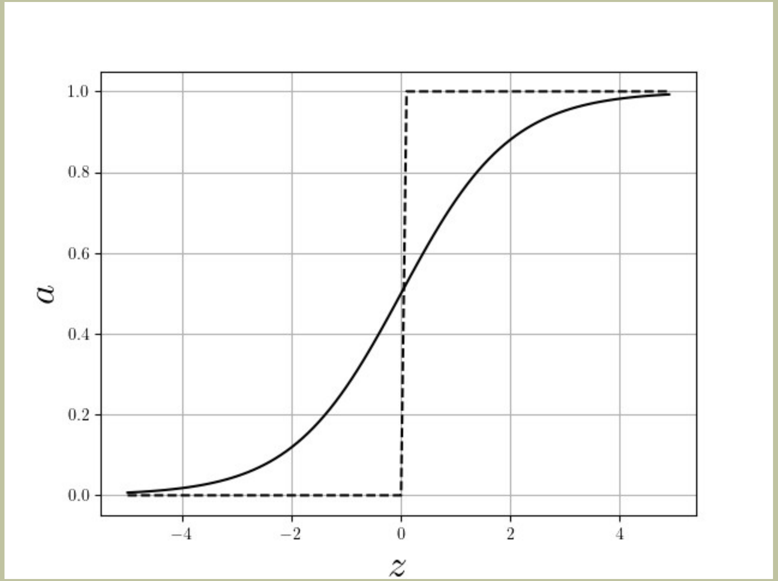


Функция активации: $\boldsymbol{a}(\boldsymbol{z})$



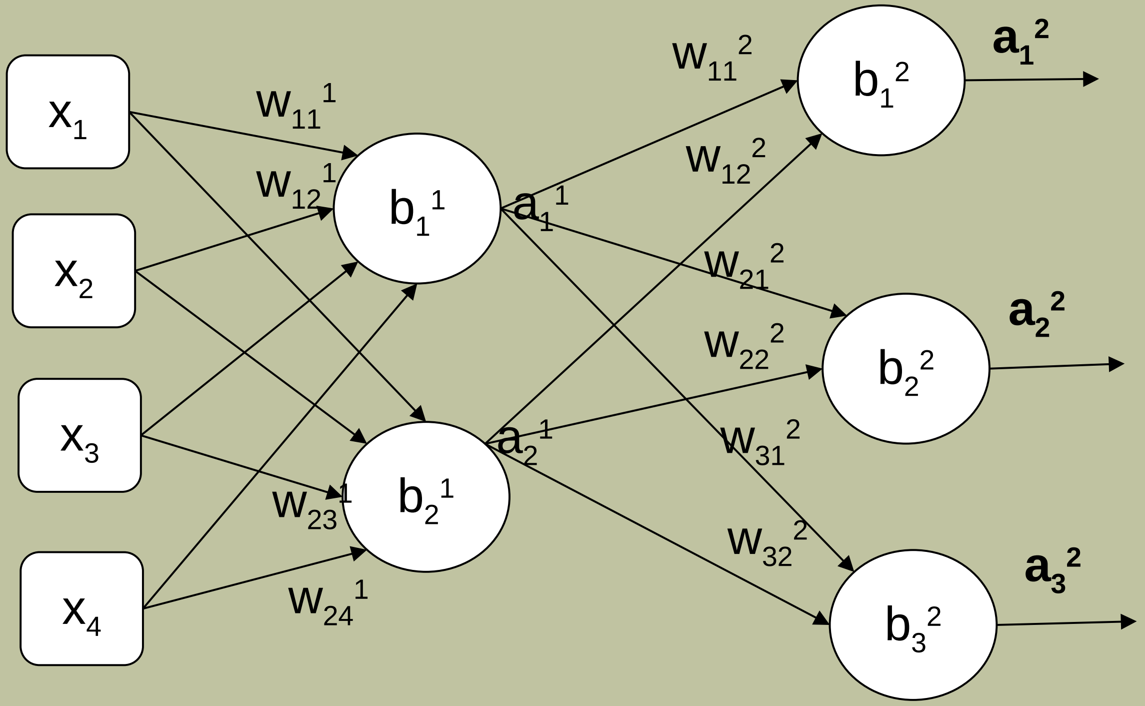
$x_1, x_2, x_3 \in R$

$z = x_1 w_1 + x_2 w_2 + x_2 w_2 + b$



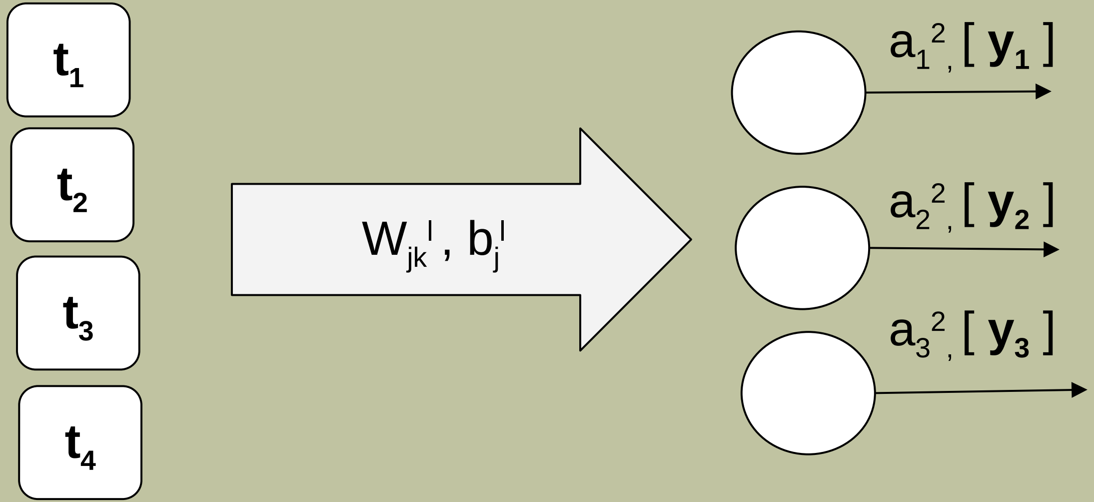
$$a(z) = \sigma(z) = \frac{1}{1+e^{-z}}$$

Нейронные сети: веса, смещения, слои.

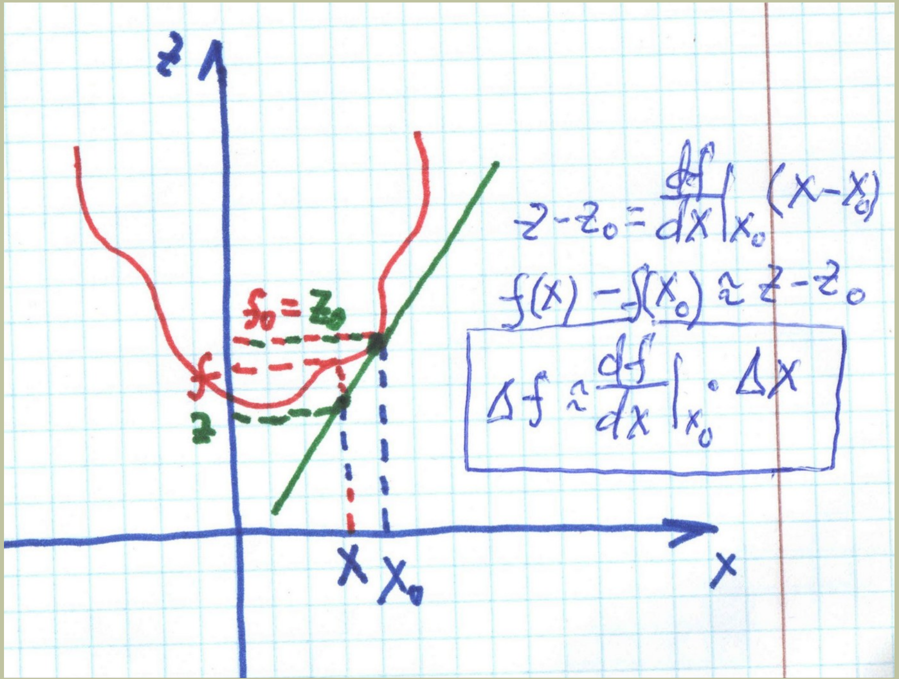


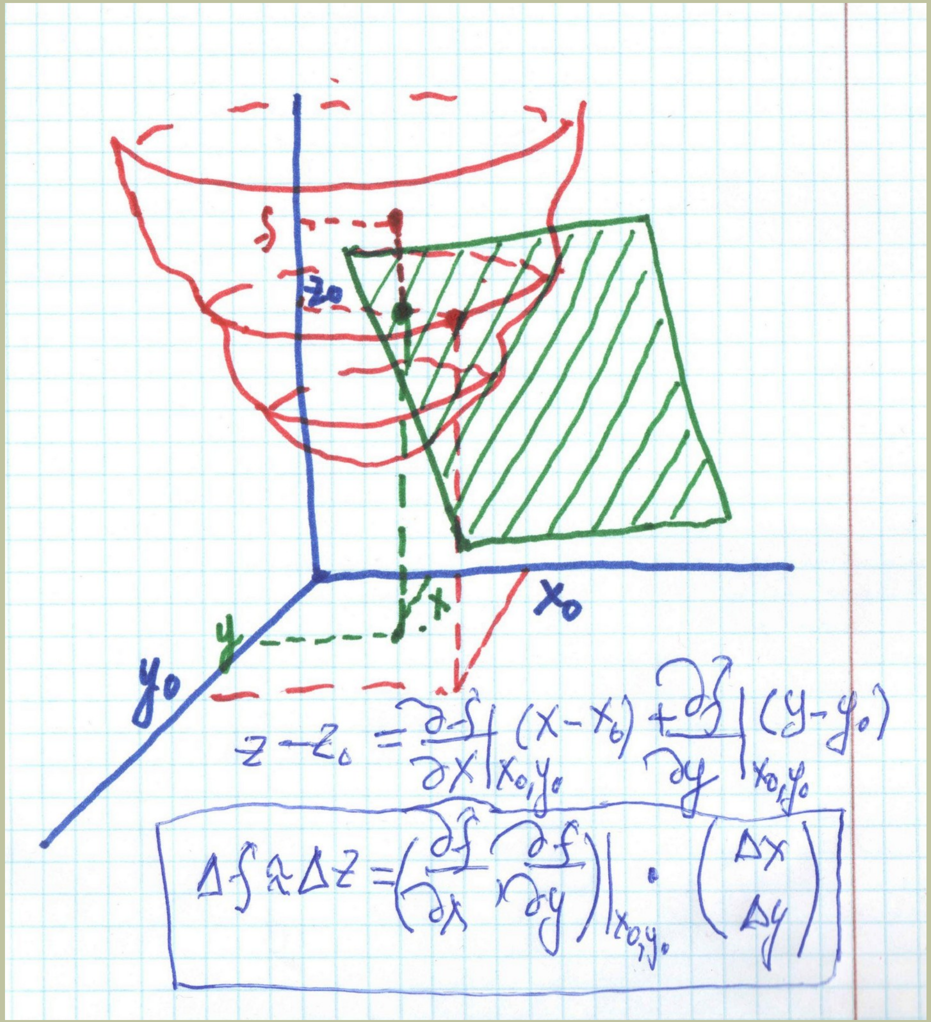
$$a_j^l = \sigma\left(\sum_k w_{jk}^l a_k^{l-1} + b_j^l\right)$$

Обучение НС: обучающие данные, функция потерь (затрат), [стохастический] градиентный спуск.



$$C(w,b) = \sum_t \|y(t) - a\|^2 = \sum_t \sum_j (y_j(t) - a_j^L)^2$$





$$\Delta x = -\varepsilon \frac{\partial f}{\partial x}, \ \Delta y = -\varepsilon \frac{\partial f}{\partial y}$$

$$\Delta f = -\varepsilon \left[\left(\frac{\partial f}{\partial x}\right)^2 + \left(\frac{\partial f}{\partial y}\right)^2 \right]$$

Обучение НС: алгоритм обратного распространения ошибки.

$$\delta^l_j \equiv \frac{\partial C}{\partial z^l_j}$$

$$\delta^L_j = \frac{\partial C}{\partial z^L_j} \qquad \text{(I)}$$

$$\delta^l_j = \sum_k w^{l+1}_{jk}\delta^{l+1}_k \sigma'(z^l_j) \qquad \text{(II)}$$

$$\frac{\partial C}{\partial b^l_j} = \delta^l_j, \qquad \text{(III)}$$

$$\frac{\partial C}{\partial w^l_{jk}} = a^{l-1}_k \delta^l_j \qquad \text{(IV)}$$

$$\delta^l_j = \frac{\partial C}{\partial z^l_j} = \sum_k \frac{\partial C}{\partial z^{l+1}_k}\frac{\partial z^{l+1}_k}{\partial z^l_j} =$$

$$= \sum_k \frac{\partial z^{l+1}_k}{\partial z^l_j}\cdot \delta^{l+1}_k$$

$$z^{l+1}_k = \sum_j \omega^{l+1}_{kj}\cdot a^l_j + b^{l+1}_k =$$

$$= \sum_j \omega^{l+1}_{kj}\,\sigma(z^l_j) + b^{l+1}_k \Rightarrow$$

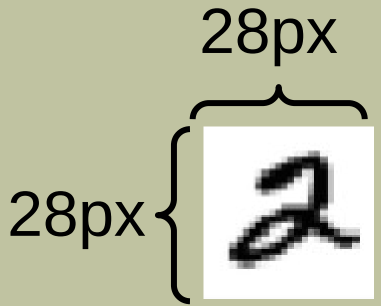
$$\frac{\partial z^{l+1}_k}{\partial z^l_j} = \omega^{l+1}_{kj}\,\sigma'(z^l_j) \Rightarrow$$

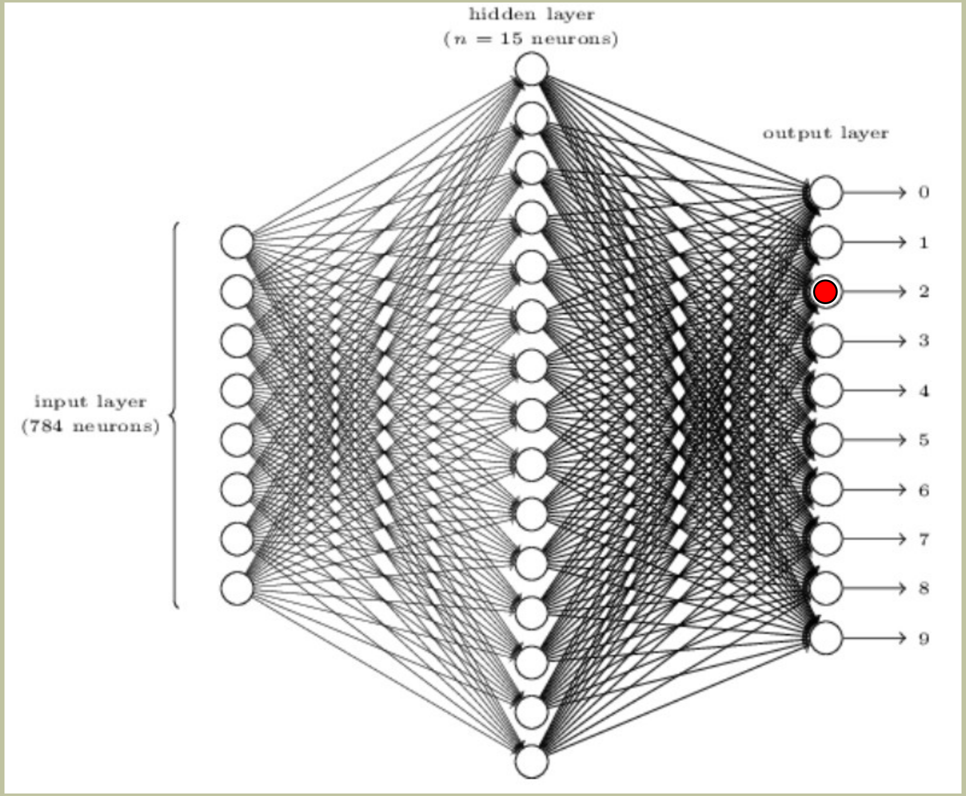
$$\boxed{\delta^l_j = \sum_k \omega^{l+1}_{kj}\,\delta^{l+1}_k\,\sigma'(z^l_j)}$$

Итак:

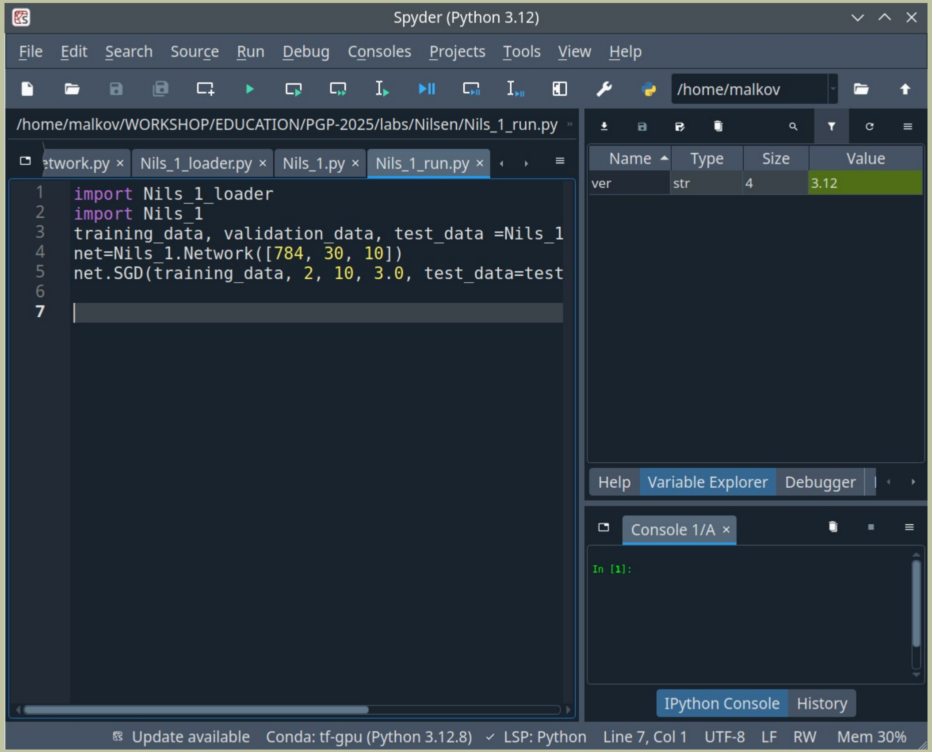
1. Задать активацию входного слоя
2. Прямое прохождение
3. Выходная ошибка
4. Обратное распространение ошибки
5. Вычисление градиента функции стоимости
6. Коррекция весов и смещений

Пример: сеть для распознавания цифр,
Michael Nielsen, http://neuralnetworksanddeeplearning.com/index.html





malkov@192:~/> **conda create --name tf-gpu python=3.12 tensorflow-gpu**
malkov@192:~/PGP-2025/labs/Nilsen> **conda activate tf-gpu**
(tf-gpu) malkov@192:~/PGP-2025/labs/Nilsen> **spyder &**



[*conda create --name* ***tf-gpu2*** *python=2.7 tensorflow-gpu*]

Инициализация архитектуры сети, весов и смещений

_Nils_1_run.py_

```
import Nils_1
net=Nils_1.Network([784, 30, 10])
………………………………………………………………………………………….
```

_Nils_1.py_

```
class Network(object):
    def __init__(self, sizes):
        self.num_layers = len(sizes)
        self.sizes = sizes
        self.biases = [np.random.randn(y, 1) for y in sizes[1:]]
        self.weights = [np.random.randn(y, x)
                        for x, y in zip(sizes[:-1], sizes[1:])]
…………………………………………………………………………………………..
```

Загрузка данных для обучения

```
import Nils_1_loader
training_data, validation_data, test_data =Nils_1_loader.load_data_wrapper()
……………………………………………………………………………………...
```

https://www.kaggle.com/datasets/pablotab/mnistpklgz

Nils_1_loader.py

```
import pickle
import gzip

def load_data():
    f = gzip.open('../data/mnist.pkl.gz', 'rb')
    training_data, validation_data, test_data = pickle.load(f, encoding='latin1')
    f.close()
    return (training_data, validation_data, test_data)
```

training_data

Кортеж из двух элементов:

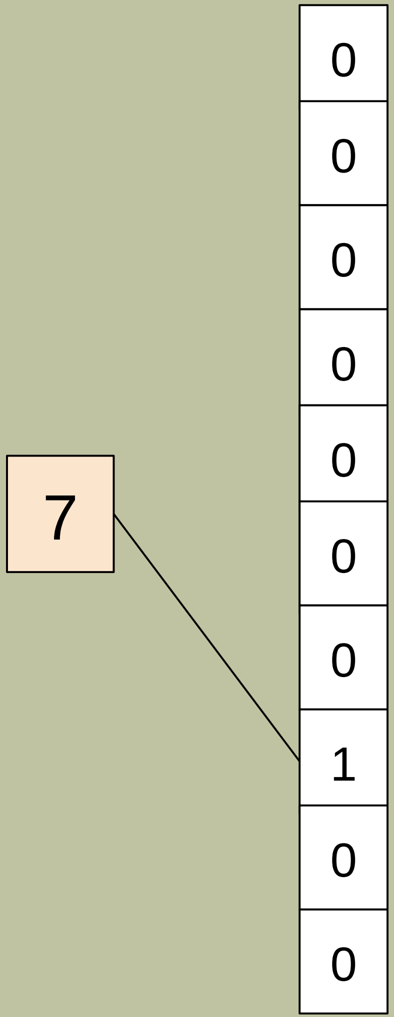
Index	Type	Size	
0	Array of float32	(50000, 784)	[[0. 0. 0. ... 0. 0. 0.]
1	Array of int64	(50000,)	[5 0 4 ... 8 4 8]

	682	683	684	685	686	687	688	689	690
0	0.238281	0.945312	0.992188	0.992188	0.203125	0	0	0	0
1	0	0	0	0	0	0	0	0	0
2	0	0	0	0	0	0	0	0	0
3	0	0	0	0	0	0	0	0	0
4	0	0	0	0	0	0	0.699219	0.257812	0
5	0	0.101562	0.613281	0.417969	0	0	0	0	0
6	0	0	0	0.238281	0.742188	0.5	0.0898438	0.0234375	0
7	0	0	0	0	0	0	0	0	0
8	0	0	0	0	0	0	0	0	0
9	0.992188	0.992188	0.535156	0	0	0	0	0	0
10	0	0	0	0	0	0	0	0	0
11	0	0	0	0	0	0	0	0	0
12	0	0	0	0.808594	0.984375	0.453125	0	0	0

	0
30	3
31	1
32	3
33	4
34	7
35	2
36	7
37	1
38	2
39	1
40	1
41	7
42	4
43	2

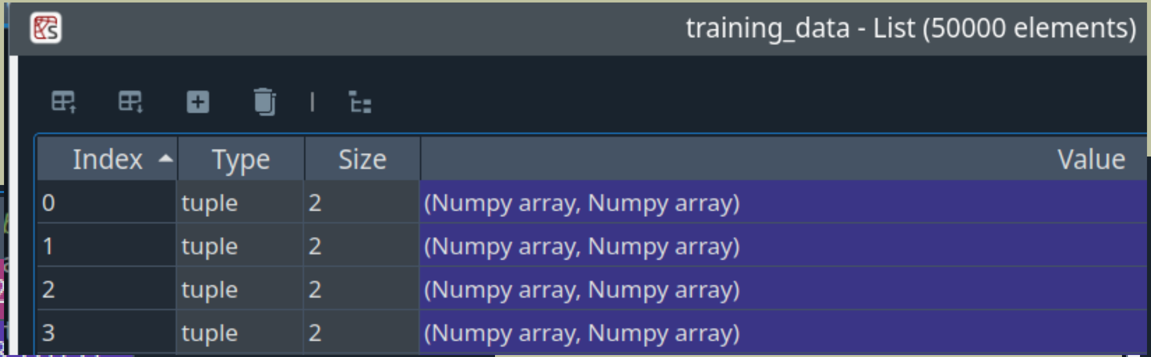
```
import numpy as np

def load_data_wrapper():
    tr_d, va_d, te_d = load_data()
    training_inputs = [np.reshape(x, (784, 1)) for x in tr_d[0]]
    training_results = [vectorized_result(y) for y in tr_d[1]]
    training_data = list(zip(training_inputs, training_results))
    validation_inputs = [np.reshape(x, (784, 1)) for x in va_d[0]]
    validation_data = list(zip(validation_inputs, va_d[1]))
    test_inputs = [np.reshape(x, (784, 1)) for x in te_d[0]]
    test_data = list(zip(test_inputs, te_d[1]))
    return (training_data, validation_data, test_data)

def vectorized_result(j):
    e = np.zeros((10, 1))
    e[j] = 1.0
    return e
```

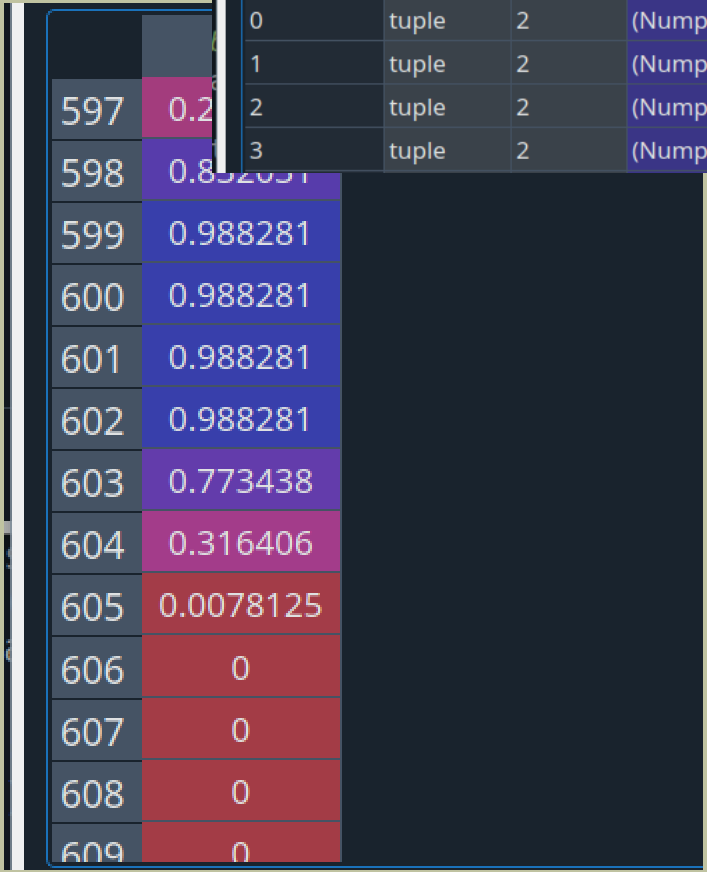
Реструктури -
рование данных

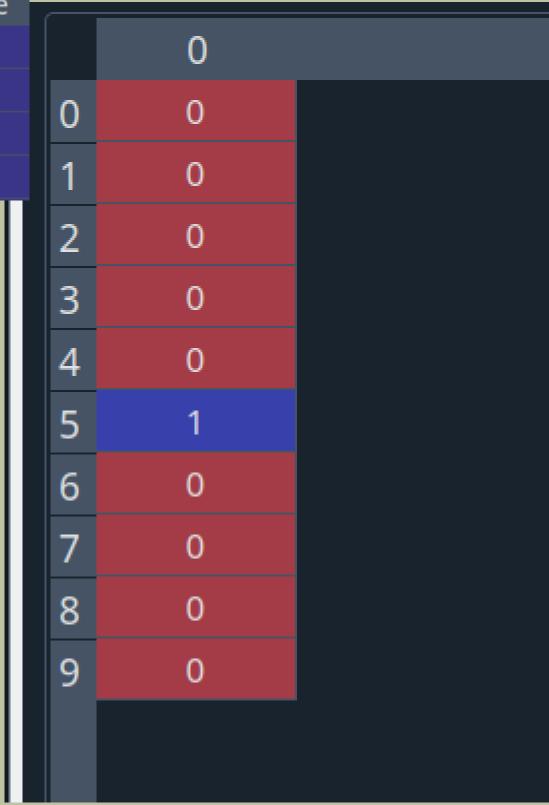


training_results = [vectorized_result(y) for y in tr_d[1]]

```
def vectorized_result(j):
    e = np.zeros((10, 1))
    e[j] = 1.0
    return e
```







После реструктурирования данных.

Реализация градиентного спуска

```
net.SGD(training_data, 20, 10, 3.0, test_data=test_data)
```

```
def SGD(self, training_data, epochs, mini_batch_size, eta, test_data=None):
    if test_data: n_test = len(test_data)
    n = len(training_data)
    for j in range(epochs):
        random.shuffle(training_data)
        mini_batches = [
            training_data[k:k+mini_batch_size]
            for k in range(0, n, mini_batch_size)]
        for mini_batch in mini_batches:
            self.update_mini_batch(mini_batch, eta)
        if test_data:
            print( "Epoch {0}: {1} / {2}".format(
                j, self.evaluate(test_data), n_test))
        else:
            print( "Epoch {0} complete".format(j))
```

Тестовый прогон

```
        if test_data:
            print( "Epoch {0}: {1} / {2}".format(
                j, self.evaluate(test_data), n_test))
        else:
            print( "Epoch {0} complete".format(j))

    def feedforward(self, a):
        for b, w in zip(self.biases, self.weights):
            a = sigmoid(np.dot(w, a)+b)
        return a

    def evaluate(self, test_data):
        test_results = [(np.argmax(self.feedforward(x)), y)
                        for (x, y) in test_data]
        return sum(int(x == y) for (x, y) in test_results)
```

```
(tf-gpu) malkov@192:~/PGP-2025/labs/Nilsen> python Nils_1_run.py
Epoch 0: 9123 / 10000
Epoch 1: 9257 / 10000
Epoch 2: 9355 / 10000
Epoch 3: 9360 / 10000
Epoch 4: 9407 / 10000
Epoch 5: 9422 / 10000
Epoch 6: 9435 / 10000
```

```python
def update_mini_batch(self, mini_batch, eta):
    nabla_b = [np.zeros(b.shape) for b in self.biases]
    nabla_w = [np.zeros(w.shape) for w in self.weights]
    for x, y in mini_batch:
        delta_nabla_b, delta_nabla_w = self.backprop(x, y)
        nabla_b = [nb+dnb for nb, dnb in zip(nabla_b, delta_nabla_b)]
        nabla_w = [nw+dnw for nw, dnw in zip(nabla_w, delta_nabla_w)]
    self.weights = [w-(eta/len(mini_batch))*nw
                    for w, nw in zip(self.weights, nabla_w)]
    self.biases = [b-(eta/len(mini_batch))*nb
                   for b, nb in zip(self.biases, nabla_b)]
```

```
def backprop(self, x, y):
    nabla_b = [np.zeros(b.shape) for b in self.biases]
    nabla_w = [np.zeros(w.shape) for w in self.weights]
    activation = x
    activations = [x]
    zs = []
    for b, w in zip(self.biases, self.weights):
        z = np.dot(w, activation)+b
        zs.append(z)
        activation = sigmoid(z)
        activations.append(activation)
    delta = self.cost_derivative(activations[-1], y) * \
        sigmoid_prime(zs[-1])
    nabla_b[-1] = delta
    nabla_w[-1] = np.dot(delta, activations[-2].transpose())
```

```
        for l in range(2, self.num_layers):
            z = zs[-l]
            sp = sigmoid_prime(z)
            delta = np.dot(self.weights[-l+1].transpose(), delta) * sp
            nabla_b[-l] = delta
            nabla_w[-l] = np.dot(delta, activations[-l-1].transpose())
        return (nabla_b, nabla_w)

    def cost_derivative(self, output_activations, y):
        return (output_activations-y)
```

```
def sigmoid(z):
    return 1.0/(1.0+np.exp(-z))
def sigmoid_prime(z):
    return sigmoid(z)*(1-sigmoid(z))
```