Министерство науки и высшего образования Российской Федерации

Федеральное государственное бюджетное образовательное учреждение высшего образования
«МОСКОВСКИЙ ГОСУДАРСТВЕННЫЙ ТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ имени Н.Э.БАУМАНА
(национальный исследовательский университет)»

Факультет: Информатика и системы управления

Кафедра: Теоретическая информатика и компьютерные технологии

Лабораторная работа №1

«Персептрон»

По дисциплине «Теория искусственных нейронных сетей»

Работу выполнил

студент группы ИУ9-72Б

Владиславов А.Г.

Москва, 2021

```
import math
import random
from PIL import Image


class Sample:
    def __init__(self, input, expected):
        self.input = input
        self.expected = expected


class Matrix:
    def __init__(self, values):
        self.values = values

    def add_matrix(self, other):
        res = get_empty_matrix(self.rows, self.columns)
        for i in range(self.rows):
            for j in range(self.columns):
                res.values[i][j] = self.values[i][j] + other.values[i]
[j]
        return res

    def multiply_matrix(self, other):
        res = get_empty_matrix(self.rows, other.columns)
        for i in range(self.rows):
            for j in range(other.columns):
                for k in range(self.columns):
                    res.values[i][j] += self.values[i][k] *
other.values[k][j]
        return res

    def multiply_vector(self, other):
        res = get_empty_vector(self.rows)
        for i in range(self.rows):
            for j in range(self.columns):
                res.values[i] += self.values[i][j] * other.values[j]
        return res

    def multiply_scalar(self, value):
        res = get_empty_matrix(self.rows, self.columns)
        for i in range(self.rows):
            for j in range(self.columns):
                res.values[i][j] = value * self.values[i][j]
        return res

    def transpose(self):
        res = get_empty_matrix(self.columns, self.rows)
        for i in range(self.rows):
```

```
            for j in range(self.columns):
                res.values[j][i] = self.values[i][j]
        return res

    @property
    def rows(self):
        return len(self.values)

    @property
    def columns(self):
        if self.rows == 0:
            return 0
        return len(self.values[0])


class Vector:
    def __init__(self, values):
        self.values = values

    def get_max_index(self):
        ans = 0
        for i in range(self.elements):
            if self.values[i] > self.values[ans]:
                ans = i
        return ans

    @property
    def length(self):
        res = 0
        for i in self.values:
            res += i**2
        return math.sqrt(res)

    def add_vector(self, other):
        res = get_empty_vector(self.elements)
        for i in range(self.elements):
            res.values[i] = self.values[i] + other.values[i]
        return res

    def hadamar_product(self, other):
        res = get_empty_vector(self.elements)
        for i in range(self.elements):
            res.values[i] = self.values[i] * other.values[i]
        return res

    def to_matrix(self):
        return Matrix([[x] for x in self.values])

    def multiply_scalar(self, value):
```

```
        return self.map(lambda x: x * value)

    def map(self, func):
        return Vector(list(map(func, self.values)))

    @property
    def elements(self):
        return len(self.values)


def get_random_vector(n, a, b):
    res = get_empty_vector(n)
    for i in range(n):
        res.values[i] = random.uniform(a, b)
    return res

def get_random_matrix(n, m, a, b):
    res = get_empty_matrix(n, m)
    for i in range(n):
        for j in range(m):
            res.values[i][j] = random.uniform(a, b)
    return res


def get_empty_matrix(n, m):
    return Matrix([[0] * m for _ in range(n)])


def get_empty_vector(n):
    return Vector([0] * n)


def sigmoid(x):
    return 1 / (1 + math.exp(-x))


def sigmoid_derivative(x):
    return sigmoid(x) * (1 - sigmoid(x))


def get_samples():
    samples = []
    for i in range(10):
        res = get_empty_vector(10)
        res.values[i] = 1
        for j in range(10):
            name = f"../datasets/digits/{i}_{j}.bmp"
            image = Image.open(name)
            vec = Vector([])
```

```
            for k in range(image.width):
                for l in range(image.height):
                    vec.values.append(image.getpixel((k, l)))
            samples.append(Sample(vec.multiply_scalar(1 / vec.length),
res))
    return samples

import math

from matplotlib import pyplot as plt


ETA = 10


EPOCHS = [10, 20, 30, 40, 50, 60, 70, 80, 90, 100]

class Prediction:
    def __init__(self, sample, prediction):
        self.sample = sample
        self.prediction = prediction


class ScoreResult:
    def __init__(self, accuracy, loss, predictions):
        self.accuracy = accuracy
        self.loss = loss
        self.predictions = predictions

    def __str__(self):
        return f"accuracy: {self.accuracy * 100}%, loss: {self.loss}"


class RunResult:
    def __init__(self, activations, weighted_inputs):
        self.activations = activations
        self.weighted_inputs = weighted_inputs


class Network:
    def __init__(self, layers):
        self.layers = layers
        self.weights = [get_empty_matrix(0, 0)]
        self.biases = [get_empty_vector(0)]
        for i in range(1, len(layers)):
            self.weights.append(get_random_matrix(layers[i], layers[i
- 1], -1, 1))
            self.biases.append(get_random_vector(layers[i], -1, 1))
```

```
    def train(self, samples, epochs):
        for epoch in range(epochs):
            print(f"{epoch + 1}/{epochs}")
            errors = []
            activations = []
            for sample in samples:
                result = self.run(sample)
                activations.append(result.activations)
                cur_errors = self.backprop(sample, result)
                errors.append(cur_errors)
            self.descend(samples, activations, errors)

    def run(self, sample):
        activations = [get_empty_vector(layer) for layer in
self.layers]
        activations[0] = sample.input
        weighted_inputs = [get_empty_vector(layer) for layer in
self.layers]
        for j in range(1, len(self.layers)):
            weighted_inputs[j] = (
                self.weights[j]
                .multiply_vector(activations[j - 1])
                .add_vector(self.biases[j])
            )
            activations[j] =
weighted_inputs[j].map(self.calc_activation)
        return RunResult(activations, weighted_inputs)

    def backprop(self, sample, result):
        errors = [get_empty_vector(layer) for layer in self.layers]
        nabla_cost = self.calc_nabla_cost(sample.expected,
result.activations[-1])
        errors[-1] = nabla_cost.hadamar_product(
            result.weighted_inputs[-
1].map(self.calc_activation_derivative)
        )
        for j in reversed(range(1, len(self.layers) - 1)):
            errors[j] = (
                self.weights[j + 1]
                .transpose()
                .multiply_vector(errors[j + 1])
                .hadamar_product(result.weighted_inputs[j].map(self.ca
lc_activation_derivative))
            )
        return errors

    def descend(self, samples, activations, errors):
        for i in range(1, len(self.layers)):
            acc_weights = get_empty_matrix(self.layers[i],
self.layers[i - 1])
```

```
            acc_biases = get_empty_vector(self.layers[i])
            for j in range(len(samples)):
                acc_weights = acc_weights.add_matrix(
                    errors[j][i]
                    .to_matrix()
                    .multiply_matrix(activations[j][i -
1].to_matrix().transpose())
                )
                acc_biases = acc_biases.add_vector(errors[j][i])
            factor = -ETA / len(samples)
            self.weights[i] = self.weights[i].add_matrix(
                acc_weights.multiply_scalar(factor)
            )
            self.biases[i] = self.biases[i].add_vector(
                acc_biases.multiply_scalar(factor)
            )

    def score(self, samples):
        cost = 0
        accurate = 0
        predictions = []
        for sample in samples:
            res = self.run(sample)
            out = res.activations[-1]
            pred = Prediction(sample, out)
            predictions.append(pred)
            cost += self.calc_cost(sample.expected, out)
            if out.get_max_index() == sample.expected.get_max_index():
                accurate += 1
        return ScoreResult(accurate / len(samples), cost /
len(samples), predictions)

    def calc_cost(self, expected, out):
        res = 0
        for y, a in zip(expected.values, out.values):
            res += - (y * math.log(a) + (1 - y) * math.log(1 - a))
        return res

    def calc_nabla_cost(self, expected, activations):
        res = get_empty_vector(activations.elements)
        for i in range(activations.elements):
            res.values[i] = (1 - expected.values[i]) / (1 -
activations.values[i]) - expected.values[i] / activations.values[i]
        return res

    def calc_activation(self, x):
        return sigmoid(x)

    def calc_activation_derivative(self, x):
```

```
        return sigmoid_derivative(x)


network = Network([24, 10])
samples = get_samples()
scores = []
for i, epochs in enumerate(EPOCHS):
    network.train(samples, epochs)
    scores.append(network.score(samples))

xs = EPOCHS
accuracies = []
losses = []
for i in range(len(EPOCHS)):
    y = scores[i]
    accuracies.append(y.accuracy)
    losses.append(y.loss)
    print('EPOCHS:', EPOCHS[i], y)

plt.plot(xs, accuracies)
plt.plot(xs, losses)
plt.legend(['accuracy', 'loss'])
```

```
1/10
2/10
3/10
4/10
5/10
6/10
7/10
8/10
9/10
10/10
1/20
2/20
3/20
4/20
5/20
6/20
7/20
8/20
9/20
10/20
11/20
12/20
13/20
14/20
15/20
```

16/20
17/20
18/20
19/20
20/20
1/30
2/30
3/30
4/30
5/30
6/30
7/30
8/30
9/30
10/30
11/30
12/30
13/30
14/30
15/30
16/30
17/30
18/30
19/30
20/30
21/30
22/30
23/30
24/30
25/30
26/30
27/30
28/30
29/30
30/30
1/40
2/40
3/40
4/40
5/40
6/40
7/40
8/40
9/40
10/40
11/40
12/40
13/40
14/40
15/40

16/40
17/40
18/40
19/40
20/40
21/40
22/40
23/40
24/40
25/40
26/40
27/40
28/40
29/40
30/40
31/40
32/40
33/40
34/40
35/40
36/40
37/40
38/40
39/40
40/40
1/50
2/50
3/50
4/50
5/50
6/50
7/50
8/50
9/50
10/50
11/50
12/50
13/50
14/50
15/50
16/50
17/50
18/50
19/50
20/50
21/50
22/50
23/50
24/50
25/50

26/50
27/50
28/50
29/50
30/50
31/50
32/50
33/50
34/50
35/50
36/50
37/50
38/50
39/50
40/50
41/50
42/50
43/50
44/50
45/50
46/50
47/50
48/50
49/50
50/50
1/60
2/60
3/60
4/60
5/60
6/60
7/60
8/60
9/60
10/60
11/60
12/60
13/60
14/60
15/60
16/60
17/60
18/60
19/60
20/60
21/60
22/60
23/60
24/60
25/60

26/60
27/60
28/60
29/60
30/60
31/60
32/60
33/60
34/60
35/60
36/60
37/60
38/60
39/60
40/60
41/60
42/60
43/60
44/60
45/60
46/60
47/60
48/60
49/60
50/60
51/60
52/60
53/60
54/60
55/60
56/60
57/60
58/60
59/60
60/60
1/70
2/70
3/70
4/70
5/70
6/70
7/70
8/70
9/70
10/70
11/70
12/70
13/70
14/70
15/70

16/70
17/70
18/70
19/70
20/70
21/70
22/70
23/70
24/70
25/70
26/70
27/70
28/70
29/70
30/70
31/70
32/70
33/70
34/70
35/70
36/70
37/70
38/70
39/70
40/70
41/70
42/70
43/70
44/70
45/70
46/70
47/70
48/70
49/70
50/70
51/70
52/70
53/70
54/70
55/70
56/70
57/70
58/70
59/70
60/70
61/70
62/70
63/70
64/70
65/70

66/70
67/70
68/70
69/70
70/70
1/80
2/80
3/80
4/80
5/80
6/80
7/80
8/80
9/80
10/80
11/80
12/80
13/80
14/80
15/80
16/80
17/80
18/80
19/80
20/80
21/80
22/80
23/80
24/80
25/80
26/80
27/80
28/80
29/80
30/80
31/80
32/80
33/80
34/80
35/80
36/80
37/80
38/80
39/80
40/80
41/80
42/80
43/80
44/80
45/80

46/80
47/80
48/80
49/80
50/80
51/80
52/80
53/80
54/80
55/80
56/80
57/80
58/80
59/80
60/80
61/80
62/80
63/80
64/80
65/80
66/80
67/80
68/80
69/80
70/80
71/80
72/80
73/80
74/80
75/80
76/80
77/80
78/80
79/80
80/80
1/90
2/90
3/90
4/90
5/90
6/90
7/90
8/90
9/90
10/90
11/90
12/90
13/90
14/90
15/90

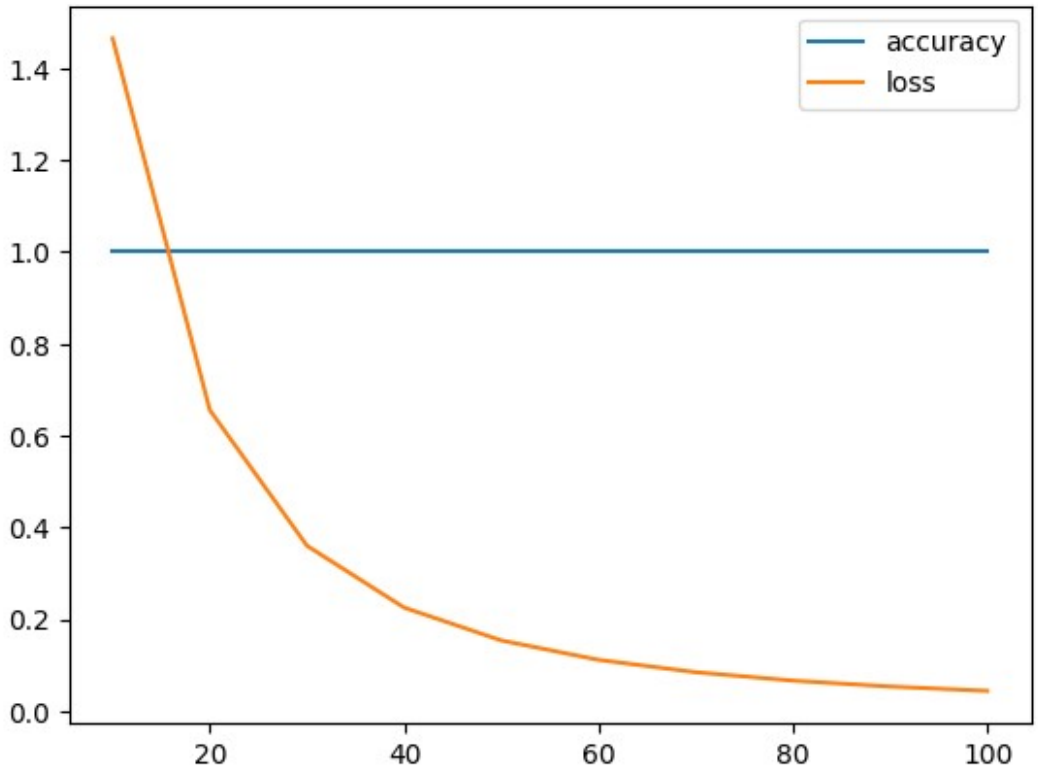
16/90
17/90
18/90
19/90
20/90
21/90
22/90
23/90
24/90
25/90
26/90
27/90
28/90
29/90
30/90
31/90
32/90
33/90
34/90
35/90
36/90
37/90
38/90
39/90
40/90
41/90
42/90
43/90
44/90
45/90
46/90
47/90
48/90
49/90
50/90
51/90
52/90
53/90
54/90
55/90
56/90
57/90
58/90
59/90
60/90
61/90
62/90
63/90
64/90
65/90

66/90
67/90
68/90
69/90
70/90
71/90
72/90
73/90
74/90
75/90
76/90
77/90
78/90
79/90
80/90
81/90
82/90
83/90
84/90
85/90
86/90
87/90
88/90
89/90
90/90
1/100
2/100
3/100
4/100
5/100
6/100
7/100
8/100
9/100
10/100
11/100
12/100
13/100
14/100
15/100
16/100
17/100
18/100
19/100
20/100
21/100
22/100
23/100
24/100
25/100

26/100
27/100
28/100
29/100
30/100
31/100
32/100
33/100
34/100
35/100
36/100
37/100
38/100
39/100
40/100
41/100
42/100
43/100
44/100
45/100
46/100
47/100
48/100
49/100
50/100
51/100
52/100
53/100
54/100
55/100
56/100
57/100
58/100
59/100
60/100
61/100
62/100
63/100
64/100
65/100
66/100
67/100
68/100
69/100
70/100
71/100
72/100
73/100
74/100
75/100

```
76/100
77/100
78/100
79/100
80/100
81/100
82/100
83/100
84/100
85/100
86/100
87/100
88/100
89/100
90/100
91/100
92/100
93/100
94/100
95/100
96/100
97/100
98/100
99/100
100/100
EPOCHS: 10 accuracy: 100.0%, loss: 1.4651041176915587
EPOCHS: 20 accuracy: 100.0%, loss: 0.655766562433141
EPOCHS: 30 accuracy: 100.0%, loss: 0.3602395882698006
EPOCHS: 40 accuracy: 100.0%, loss: 0.22557100318044082
EPOCHS: 50 accuracy: 100.0%, loss: 0.1540625577304932
EPOCHS: 60 accuracy: 100.0%, loss: 0.11182592972881826
EPOCHS: 70 accuracy: 100.0%, loss: 0.08487325448287569
EPOCHS: 80 accuracy: 100.0%, loss: 0.06664240917084355
EPOCHS: 90 accuracy: 100.0%, loss: 0.05374039825854897
EPOCHS: 100 accuracy: 100.0%, loss: 0.04427443874433164

<matplotlib.legend.Legend at 0x1173a9250>
```


accuracy
loss
1.4
1.2
1.0
0.8
0.6
0.4
0.2
0.0
20
40
60
80
100