Практическое задание №4

Вариант №2

Условие:

Необходимо реализовать нейронную сеть вычисляющую результат заданной логической операции. Затем реализовать функции, которые будут симулировать работу построенной модели. Функции должны принимать тензор входных данных и список весов. Должно быть реализовано 2 функции:

1. Функция, в которой все операции реализованы как поэлементные операции над тензорами
2. Функция, в которой все операции реализованы с использованием операций над тензорами из NumPy

Для проверки корректности работы функций необходимо:

1. Инициализировать модель и получить из нее веса (Как получить веса слоя, Как получить список слоев модели)
2. Прогнать датасет через не обученную модель и реализованные 2 функции. Сравнить результат.
3. Обучить модель и получить веса после обучения
4. Прогнать датасет через обученную модель и реализованные 2 функции. Сравнить результат.

Примечание: так как множество всех наблюдений ограничен, то обучение проводить можно на всем датасете без контроля.

Вариант 2

(a or b) xor not(b and c)

Выполнение:

Данные:

0;0;0;1

0;0;1;1

0;1;0;0

0;1;1;1

1;0;0;0

1;0;1;0

1;1;0;0

1;1;1;1

Была составлена модель ИНС с двумя слоями:

```
model = Sequential()
model.add(Dense(8, activation='relu', input_shape=(3,)))
model.add(Dense(8, activation='relu'))
model.add(Dense(1, activation='sigmoid'))
model.compile(optimizer='adam', loss='binary_crossentropy',
metrics=['accuracy'])
```

Были написаны вспомогательные функции:

Реализация relu:

```
def naive_relu(x):
    assert len(x.shape) == 2  # проверка размерности 2
    x = x.copy()  # копирования от защиты изменения исходного тензора
    for i in range(x.shape[0]):
        for j in range(x.shape[1]):
            x[i, j] = max(x[i, j], 0)
    return x
```

Реализация sigmoid:

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

Реализация скалярного произведения векторов:

```
def naive_vector_dot(x, y):
    assert len(x.shape) == 1
    assert len(y.shape) == 1
    assert x.shape[0] == y.shape[0]
    z = 0.
    for i in range(x.shape[0]):
        z += x[i] * y[i]
    return z
```

Реализация скалярного произведения матрицы и вектора:

```
def naive_matrix_matrix_dot(x, y):
    assert len(x.shape) == 2
    assert len(y.shape) == 2
    assert x.shape[1] == y.shape[0]

    z = np.zeros((x.shape[0], y.shape[1]))
    for i in range(x.shape[0]):
        for j in range(y.shape[1]):
            z[i, j] = naive_vector_dot(x[i, :], y[:, j])
    return z
```

Были написаны функции, симулирующие работу модели, где model.add(layers.Dense(16, activation='relu')) интерпретируется как output = relu(dot(W, input) + b). W – матрица весов слоя, input – входные значения, b – вектор смещения.

Релизовано две функции:

1. Функция, в которой все операции реализованы как поэлементные операции над тензорами

```
def naive_simulation(layers, input):
    lenL = len(layers) - 1;
    output = input;
    for i in range(lenL):
        output = naive_relu(naive_matrix_matrix_dot(output,
layers[i].get_weights()[0]) + layers[i].get_weights()[1])
    output = sigmoid(naive_matrix_matrix_dot(output,
layers[lenL].get_weights()[0]) + layers[lenL].get_weights()[1])
    return np.reshape(output, ((input.shape[0]), 1))
```

2. Функция, в которой все операции реализованы с использованием операций над тензорами из NumPy

```
def numpy_simulation(layers, input):
    lenL = len(layers) - 1;
    output = input;
    for i in range(lenL):
        output = np.maximum(np.dot(output, layers[i].get_weights()[0])
+ layers[i].get_weights()[1], 0)
    output = sigmoid(np.dot(output, layers[lenL].get_weights()[0]) +
layers[lenL].get_weights()[1])
    return np.reshape(output, ((input.shape[0]), 1))
```

Результаты на необученной модели:

Model	Numpy	Naive
[[0.5]	[[0.5]	[[0.5]
[0.5318045]	[0.53180452]	[0.53180452]
[0.5535644]	[0.55356445]	[0.55356445]
[0.52682245]	[0.52682248]	[0.52682248]
[0.5317482]	[0.53174818]	[0.53174818]
[0.53744256]	[0.53744254]	[0.53744254]
[0.55973285]	[0.55973286]	[0.55973286]
[0.5596192]]	[0.5596192]]	[0.5596192]]

Результаты после обучение модели:

Model	Numpy	Naive
[[0.5428818]	[[0.54288177]	[[0.54288177]
[0.70968294]	[0.70968295]	[0.70968295]
[0.4644915]	[0.46449147]	[0.46449147]

[0.7189344]	[0.71893444]	[0.71893444]
[0.44459328]	[0.44459327]	[0.44459327]
[0.5538127]	[0.55381266]	[0.55381266]
[0.4578365]	[0.4578365]	[0.4578365]
[0.62310416]]	[0.62310415]]	[0.62310415]]

Можно сделать вывод, что различия в результатах минимальны. Полученные результаты разнятся с 7 разряда только у модели, у реализации numpy и naive значения полностью совпадают.