Практическое задание 4

Аверина Ольга, 8383

Задание.

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:

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

Для проверки корректности работы функций необходимо:

- Инициализировать модель и получить из нее веса.
- Прогнать датасет через не обученную модель и реализованные 2 функции. Сравнить результат.
- Обучить модель и получить веса после обучения
- Прогнать датасет через обученную модель и реализованные 2 функции. Сравнить результат.

Вариант 1

(a **and** b) **or** (a **and** c)

Выполнение работы.

Был создан файл input.csv, в котором содержится датасет:

0,0,0

0,0,1

0,1,0

0,1,1

1,0,0

1,0,1

1,1,0

1,1,1

Вектор train_label получается путем применения к матрице исходных данных логического выражения (a **and** b) **or** (a **and** c).

```
train_label = np.asarray([logic_func(x) for x in train_data])
```

Была создана модель, состоящая из трех слоев:

```
model = models.Sequential()
model.add(layers.Dense(8, activation='relu', input_shape=(3,)))
model.add(layers.Dense(16, activation='relu'))
model.add(layers.Dense(1, activation='sigmoid'))
```

На первом скрытом слое 8 нейронов, на втором - 16, функция активации - Relu. На выходном слое функция активации - sigmoid.

Были реализованы 2 функции:

Функция, в которой все операции реализованы как поэлементные операции над тензорами. К первым двум слоям применяется функция активации naive_relu, которая применяет операцию max(0, x) к каждому элементу, на вход подается сумма произведения матрицы предыдущего слоя и весов и смещения. Затем к последнему слою применяется функция sigmoid, которая применяет к каждому элементу 1 / (1 + np.exp(-x)), входные данные те же.

```
def naive_simulation(layers, data):
```

```
        data = naive_relu(
                naive_matrix_matrix_dot(data, layers[0].get_weights()[0]) +
      layers[0].get_weights()[1])
            data = naive_relu(
                naive_matrix_matrix_dot(data, layers[1].get_weights()[0]) +
      layers[1].get_weights()[1])
            data = sigmoid(
                naive_matrix_matrix_dot(data, layers[2].get_weights()[0]) +
      layers[2].get_weights()[1])
            return data

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]):
                    row_x = x[i, :]
                    column_y = y[:, j]
                    z[i, j] = naive_vector_dot(row_x, column_y)
            return z

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

def naive_relu(x):

      assert len(x.shape) == 2
            x = x.copy()
            for i in range(x.shape[0]):
                for j in range(x.shape[1]):
                    x[i, j] = max(x[i, j], 0)
            return x
```

Функция, в которой все операции реализованы с использованием операций над тензорами из NumPy. В отличии от предыдущей функции, в данной вместо naive_relu применяется функция np.maximum из библиотеки Numpy.

```
def numpy_simulation(layers, data):

      data = np.maximum(np.dot(data, layers[0].get_weights()[0]) +
      layers[0].get_weights()[1], 0)
            data = np.maximum(np.dot(data, layers[1].get_weights()[0]) +
      layers[1].get_weights()[1], 0)
            data = sigmoid(np.dot(data, layers[2].get_weights()[0]) +
      layers[2].get_weights()[1])
            return data
```

Результат прогона датасета через необученную нейросеть:

```
Before training:
Model.predict:
[[0.5       ]
[0.5423657 ]
[0.57119715]
[0.6005243 ]
[0.49939874]
[0.512331  ]
[0.46801478]
[0.4986378 ]]
Naive:
[[0.5       ]
[0.54236568]
[0.57119715]
[0.60052429]
[0.49939874]
[0.51233102]
[0.46801478]
[0.49863779]]
Numpy:
[[0.5       ]
[0.54236568]
[0.57119715]
[0.60052429]
[0.49939874]
[0.51233102]
[0.46801478]
```

```
[0.49863779]]
```

Результаты идентичны, учитывая округление в функции model.predict().

Было проведено обучение нейросети за 700 эпох:

```
H = model.fit(train_data, train_label, epochs=700, verbose=False)
```

Результаты после обучения нейросети:

```
After training:
model.predict:
[[0.00378054]
[0.00352073]
[0.00212592]
[0.00782421]
[0.01965138]
[0.9867363 ]
[0.98822   ]
[0.99919564]]
Naive:
[[0.00378051]
[0.00352077]
[0.00212598]
[0.00782426]
[0.01965138]
[0.98673626]
[0.98821998]
[0.9991956 ]]
Numpy:
[[0.00378051]
[0.00352077]
[0.00212598]
[0.00782426]
[0.01965138]
```

```
[0.98673626]
[0.98821998]
[0.9991956 ]]
```

Сравним полученные результаты с train_label:

[0. 0. 0. 0. 0. 1. 1. 1.]

Все три результата при округлении совпадут с правильными значениями. При этом между собой они практически совпадают.