Шишкин Иван, гр. 8383

Вариант 8

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

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

Выполнение работы.

Был создан датасет:

```
train_data = np.asarray([[0, 0, 0], [0, 0, 1], [0, 1, 0], [0, 1, 1], [1, 0, 0], [1, 0, 1], [1, 1, 0], [1, 1, 1]])
```

Была создана модель, состоящая двух скрытых слоев:

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

На первом скрытом слое 16 нейронов, на втором – 8, на обоих функция активации ReLu. На выходном слое функция активации – sigmoid.

Параметр train_label (значения, которые должны получиться) задается следующим образом:

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

В нем хранятся такие значения:

[True True False False True True True False]

Для задания таких значений используется функция с логической операцией (a and c and b) xor (a or not b):

```
def my_logic_operation(x):
    return (x[0] and x[2] and x[1]) != (x[0] or not x[1])
```

Были реализованы 2 функции: функция, в которой все операции реализованы как поэлементные операции над тензорами и функция, в которой все операции реализованы с использованием операций над тензорами из NumPy.

Первая функция выглядит следующим образом:

```
def naive_simulation(model_layers, train_data_f, numOfLayers):
    for i in range(numOfLayers - 1):
        train_data_f = naive_relu(naive_matrix_matrix_dot(train_data_f,
model_layers[i].get_weights()[0])
                                  + model_layers[i].get_weights()[1])
    train_data_f    =    sigmoid(naive_matrix_matrix_dot(train_data_f,
model_layers[numOfLayers - 1].get_weights()[0])
                           +         model_layers[numOfLayers         -
1].get_weights()[1])

    return train_data_f
```

В ней программа циклом проходится по двум скрытым слоям. С помощью функции `naive_matrix_matrix_dot` производится умножение весов на значение предыдущего нейрона. Затем, к этому значению прибавляется смещение, и функцией naive_relu берется значение max(0, x). Потом цикл заканчивается и остается выходной слой. Для него проделываем все то же самое, только вместо ReLu используется сигмоидная функция.

Функция naive_relu:

```
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 sigmoid:

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

Функция naive_matrix_matrix_dot:

```
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
```

Вторая функция, в которой все операции реализованы с использованием операций над тензорами из NumPy, по сути, такая же, как и предыдущая:

```
def numpy_simulation(model_layers, train_data_f, numOfLayers):
    for i in range(numOfLayers - 1):
        train_data_f         =         np.maximum(np.dot(train_data_f,
model_layers[i].get_weights()[0])
                                  +   model_layers[i].get_weights()[1],
0)
    train_data_f             =             sigmoid(np.dot(train_data_f,
model_layers[numOfLayers - 1].get_weights()[0])
                            +         model_layers[numOfLayers         -
1].get_weights()[1])

    return train_data_f
```

Прогон датасета через необученную сеть:

Before training:

model.predict:

[[0.5]
[0.5246774]
[0.56915855]
[0.6035555]
[0.5744323]
[0.6313592]
[0.5936725]
[0.6633099]]

Naive:

[[0.5]
[0.52467738]
[0.56915855]
[0.6035555]
[0.57443231]
[0.63135924]
[0.59367255]
[0.66330991]]

Numpy:

[[0.5]
[0.52467738]
[0.56915855]
[0.6035555]
[0.57443231]
[0.63135924]
[0.59367255]
[0.66330991]]

Видно, что значения во всех трех случаях практически идентичны.

Совпадение полученных данных с результатом (True – совпало):

Naive coincidence before:

[False]

[True]

[False]

[False]

[True]

[True]

[True]

[False]

Numpy coincidence before:

[False]

[True]

[False]

[False]

[True]

[True]

[True]

[False]

До обучения сеть угадала 50% ответов.

Обучим сеть (800 эпох):

```
model.compile(optimizer='adam',              loss='binary_crossentropy',
metrics=['accuracy'])
H = model.fit(train_data, train_label, epochs=800, verbose=False)
```

Получаются следующие значения:

After training:

model.predict:

[[9.96919155e-01]

[9.96478796e-01]

[1.15177035e-02]

[6.90251589e-04]

[9.99954820e-01]

[9.99683499e-01]

[9.91880894e-01]

[6.67816401e-03]]

Naive:

[[9.96919222e-01]

[9.96478840e-01]

[1.15177402e-02]

[6.90223007e-04]

[9.99954924e-01]

[9.99683427e-01]

[9.91880892e-01]

[6.67815040e-03]]

Numpy:

[[9.96919222e-01]

[9.96478840e-01]

[1.15177402e-02]

[6.90223007e-04]

[9.99954924e-01]

[9.99683427e-01]

[9.91880892e-01]

[6.67815040e-03]]

Проверим совпадение:

Naive coincidence after:

[True]

[True]

[True]

[True]

[True]

[True]

[True]

[True]

Numpy coincidence after:

[True]

[True]

[True]

[True]

[True]

[True]

[True]

[True]

Все значения совпали. Погрешность между результатами во всех трех случаях незначительная.