Simple Neural Network in Python

1. General overview

In this project, it is investigated how things such as padding technique, learning coefficient, and the number of layers are corresponding to the learning efficiency (averaged efficiencies of 5 runs) of the neural network. A neural network was written using only the „numpy” module. Using it, data for training and testing is loaded from two different folders. Data from the AudioMNIST database is used.

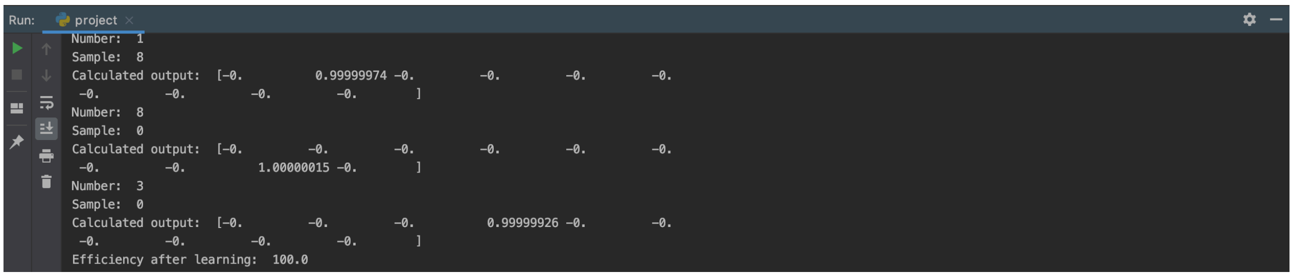
Network parameters	Number of epochs	Efficiency after learning, same data as in training	Efficiency after learning, new data
Activation function: ReLu	10	19%	11%
Layers: 1 (input - 10)	50	52%	12%
Testing dataset size: 100	500	97%	14%
	5000	100%	22%
	50000		20%

1 layer neural network

Two different padding techniques were used. The first one from https://bitbucket.org/wielgosz-info/audio-snn-unico/src/master/src/audio_snn_unico/preprocessing/ consisted of adding zeroes one both sides of the signal (so the input array can always have 8000 numbers). The other technique that was added, was conventional zero-padding, adding zeroes to the right of the array so that it meets the length requirement. Differences were negligible, but the technique from the bitbucket repository by dr Wielgosz worked slightly better, eg. 100% efficiency for the same data used in train and test sequence was achieved faster - for 500 epochs in comparison to 550 epochs using my zero-padding. Additionally, the network was tested for three different learning coefficients: 0.06, 0.6, and 6. The best performance was observed for 0.6 coefficient, then 6, and the worst one for 0.06. Nevertheless, a simple one-layer neural network failed to provide effective spoken digit recognition of unknown data, no matter the training dataset size(100, 500, 1000 samples) and other parameters. The highest achieved efficiency in recognition of new data was around 24%, which is well below the acceptance level.

Network parameters	Number of epochs	Efficiency after learning, same data as in training	Efficiency after learning, new data
Activation function: ReLu	10	16%	15%
Layers: 2 (input - 100 - 10)	50	38%	12%
	500	100%	21%
	5000	100%	23%
	50000		40%

2 layers neural network



```
Number:  1
Sample:  8
Calculated output:  [-0.          0.99999974 -0.          -0.          -0.          -0.
 -0.          -0.          -0.          -0.        ]
Number:  8
Sample:  0
Calculated output:  [-0.          -0.          -0.          -0.          -0.          -0.
 -0.          -0.          1.00000015 -0.        ]
Number:  3
Sample:  0
Calculated output:  [-0.          -0.          -0.          0.99999926 -0.          -0.
 -0.          -0.          -0.          -0.        ]
Efficiency after learning:  100.0
```

Examplary console output

The neural network consisting of two layers achieved slightly better efficiency. The middle layer with 100 neurons was added. The network learning was learning faster, and the numbers of recognized samples grew to 40% after 50000 epochs. It is still very low, but it insists that the multiple-layer network might be a solution to this problem.

2. Code evaluation

Weights initialization:

```
def init_weights(number_of_inputs, number_of_neurons):
    weights = 0.2*np.random.rand(number_of_inputs, number_of_neurons)-0.1
    return weights
```

Function compute matrix of weights of given size using np.random.rand().

Output computation:

```
def compute_outputs(weights, input_data):  # beta = 5
    u = np.matmul(weights.transpose(), input_data.transpose())
    y = u * (u>0)
    #y = 1/(1+np.exp(-0.1*u))
    return y
```

This function takes the weights matrix and input to the network and computing the output, in this case, the ReLu activation function is used, sigmoid unction is commented.

Training:

```
def train(weights_before, train_data, epochs):
    desired_outputs = np.eye(10)
    learning_coeff = 0.6
    weights_learning = weights_before
    print(weights_before)
    for i in range(epochs):
        input_digit = random.randint(0, 9)
        input_digit_sample = random.randint(0, 249)
        input_learning = train_data[input_digit][input_digit_sample]
        output_of_network = compute_outputs(weights_learning, input_learning)
        output_error = desired_outputs[input_digit]-output_of_network
        input_learning = input_learning.reshape(1, 8000)
        output_error = output_error.reshape(1, 10)
        weights_adjust = learning_coeff*np.matmul(input_learning.transpose(), output_error)
        weights_learning = weights_learning + weights_adjust
    weights_after_learning = weights_learning
    return weights_after_learning
```

The training function takes random weights initialized at the beginning, training data, and epochs. Two numbers are drawn at each iteration: one from range 0-9 and the other from 0-<number of samples>. The first one denotes which number will be chosen and the latter picks on of many samples. Then adjustment to the weights is computed using a simple algorithm.

Preparing data:

```python
def write_wav_into_list_test_or_train(path):
    x0test = []
    x1test = []
    x2test = []
    x3test = []
    x4test = []
    x5test = []
    x6test = []
    x7test = []
    x8test = []
    x9test = []
    x_test = []
    for filename in os.listdir(test_path):
        x, s = librosa.load(test_path+filename, sr=8000)
        if len(x) > 8000:
            raise ValueError("data length cannot exceed padding length.")
        elif len(x) < 8000:
            option = "padding_left_right"
            #option = "padding_only_right"
            if option == "padding_left_right":
                data = np.zeros(8000)
                offset = np.random.randint(low=0, high=8000 - len(x))
                data[offset:offset + len(x)] = x
                x = data
            else:
                data_size = len(x)
            x = np.pad(x, (0, 8000 - data_size), 'constant')
        elif len(x) == 8000:
            # nothing to do here
            x = x
        #x = ((x - np.min(x)) / np.ptp(x))/10  # 0.0-0.1 normalization
        if filename[0] == "0":
            x0test.append(x)
        elif filename[0] == "1":
            x1test.append(x)
        elif filename[0] == "2":
            x2test.append(x)
        elif filename[0] == "3":
            x3test.append(x)
        elif filename[0] == "4":
            x4test.append(x)
        elif filename[0] == "5":
            x5test.append(x)
        elif filename[0] == "6":
            x6test.append(x)
        elif filename[0] == "7":
            x7test.append(x)
        elif filename[0] == "8":
            x8test.append(x)
        else:
            x9test.append(x)
        x_test = [x0test, x1test, x2test, x3test, x4test, x5test, x6test, x7test, x8test, x9test]
    return x_test
```

Audio files in the .wav format are written into an array using the „librosa" module. Firstly, signals are downsampled to 8 kHz and then zero-padded. There are two zero-padding techniques to chose from. The first one padds the signal only on the right side, and the other on both sides. The data is stored in one big matrix in which every of ten rows consists of numerous samples of the corresponding number, eg. row 0 has many samples of zero number.

Efficiency computing:

```
def check(weights_after_learning, test_data, n):
    count = 0
    for i in range(n):
        input_digit_test = random.randint(0, 9)
        input_digit_sample_test = random.randint(0, 249)
        x = compute_outputs(weights_after_learning, test_data[input_digit_test]
[input_digit_sample_test])
        print("Number: ", input_digit_test)
        print("Sample: ", input_digit_sample_test)
        print("Calculated output: ", x)
        if x[input_digit_test] == max(x):
            count = count + 1
        else:
            count = count
    return (count/n)*100
```

The check function takes weights after learning and the other data used for testing. The variable „count" is initialized and each iteration, if a maximum element of the output array corresponds to the drawn number, it is increased by 1. The function returns efficiency in percents.

3. Future modifications

I would like to add the possibility to build multiple hidden-layer structures, but first, some code optimization is needed as various operations are quite time-consuming. Moreover, adding a simple user interface in which activation functions and other parameters can be set would be nice. And last, but not least the efficiency of the network, being the key factor, has to be increased.