ITMO UNIVERSITY

UNIVERSITY OF INFORMATION TECHNOLOGIES,
MECHANICS AND OPTICS

HOMEWORK №1

"Study of the influence of the parameters of the training sample and architecture models on the probability of handwritten digits recognition artificial neural network"

Student: Fierro Nunez Jazmin Cristina
Group: B3316
Subject: Applied Mathematics

Saint-Petersburg
2019

Homework: Implementation of a Perceptron with backpropagation

1) Goal:
To get familiarized with the implementation of the method of generalized decision function through the realization of a backpropagation perceptron.

2) Task completion:
I) Algorithm's contents:
a. Implementation of helper functions:
- Data loader (CSV loader)
- String column to float.
- String column to integer.
- Find the minimum and maximum value of each column.
- Re-scale data set columns to the range 0 – 1.
- Split the data set into k folds (epochs).

b. Accuracy calculation:
- Calculate the accuracy percentage.

c. Algorithm evaluation:
- Evaluate the algorithm using a cross validation split.

d. Neuron (perceptron) activation:
- Calculate neuron activation for an input.
- Transfer neuron activation.

e. Forward propagation:
- Forward propagate input to a network output.
- Calculate the derivative of a neuron output.

f. Backpropagation:
- Back propagate error and store in neurons.
- Update network weights with error.
- Backpropagation with Stochastic Gradient Descent.

g. Training:
- Train a network for a fixed number of epochs.
- initialize network.
- Prediction.

II) Description:

i. Initializing a network:

Given that each input connection has a respective weight, and each respective weight has its own bias, a dictionary was used to represent each neuron and store properties by names such as ‘weights’ for the weights.

Each row from the data set represents an input, and in this case, the first layer is the hidden layer. This is followed by the output layer that has one neuron for each class value.
The layers were organized as arrays of dictionaries, and the whole network is treated as an array of layers.
The initial network weights were initialized with random values from 0 to 1.

The function *initialize_network()* creates a new neural network ready for training. The function accepts three parameters: the number of inputs, the number of neurons in the hidden layer, and the number of outputs.

```
def initialize_network(n_inputs, n_hidden, n_outputs):
    network = list()
    hidden_layer = [{'weights':[random() for i in range(n_inputs + 1)]} for i in range(n_hidden)]
    network.append(hidden_layer)
    output_layer = [{'weights':[random() for i in range(n_hidden + 1)]} for i in range(n_outputs)]
    network.append(output_layer)
    return network
```

For the hidden layer, *n_hidden* neurons were created, each containing *n_inputs+1* weights, one for each input column in the data set.
The output layer that connects to the hidden layer has *n_outputs* neurons, each with *n_hidden+1* weights – each neuron in the output layer has a weight for each neuron in the hidden layer.

ii. Forward propagation:

-Neuron Propagation:

Neuron activation was calculated as the weighed sum of the inputs – alike linear regression.
The implementation was done as follows:
a*ctivation = sum(weight_i * input_i) + bias* , where *bias* will always be 1.

```
def activate(weights, inputs):
    activation = weights[-1]
    for i in range(len(weights)-1):
        activation += weights[i] * inputs[i]
    return activation
```

-Neuron transfer:

The transferring was done through Euler’s number, as follows:
output = 1 / (1+e^(-activation))

```
def transfer(activation):
    return 1.0 / (1.0 + exp(-activation))
```

-Forward Propagation:

A neuron's input value is stored in the neuron with the name 'output', respectively. The outputs are collected in an array named *new_inputs* that will be copied into the original array *inputs* to be used for the following layer.

```
def forward_propagate(network, row):
    inputs = row
    for layer in network:
        new_inputs = []
        for neuron in layer:
            activation = activate(neuron['weights'], inputs)
            neuron['output'] = transfer(activation)
            new_inputs.append(neuron['output'])
        inputs = new_inputs
    return inputs
```

iii. Error backpropagation:

- Transfer derivative:

Given an output value, a slope needs to be calculated. To do this, a sigmoid transfer function was implemented:

*derivative = output * (1.0 – output)*

-Error backpropagation:

This was calculated in order to get the error input to propagate backwards through the network. The error for a neuron can be calculated as follows:

*error = (expected – output) * transfer_derivative(output)*

Where *expected* is the expected output value for the neuron, *output* is the output value for the neuron and *transfer_derivative()* calculates the slope of the neuron's output value.

This calculation is used for the neurons in the output layer, and the expected value is the class value itself.

The back propagated error is accumulated after having passed through biased weights in order to determine the error for hidden layered neurons, as follows:

*error = (weight_k * error_j) * transfer_derivative(output)*

Where *error_j* is the error signal from the j-th neuron in the output layer, *weight_k* is the weight that connects the k-th neuron to the current neuron.

This procedure was implemented as follows:

```
def backprop_err(network, expected):
    for i in reversed(range(len(network))):
        layer = network[i]
        errors = list()
        if i != len(network)-1:
            for j in range(len(layer)):
                error = 0.0
                for neuron in network[i + 1]:
                    error += (neuron['weights'][j] * neuron['delta'])
                errors.append(error)
        else:
            for j in range(len(layer)):
                neuron = layer[j]
                errors.append(expected[j] - neuron['output'])
        for j in range(len(layer)):
            neuron = layer[j]
            neuron['delta'] = errors[j] * transfer_derivative(neuron['output'])
```

The signal for each neuron is stored as 'delta' (as of Δ). The layers of the network are iterated in reverse order, ensuring that the neurons have the 'delta' values calculated first so that the neurons in the hidden layers can use in the subsequent iteration.

iv. Training:
The network is trained with the help of a stochastic gradient descent algorithm.

-Update weights:
Once errors are calculated through back propagation, the network weights are updated as follows:
*weight = weight + learning_rate * error * input*

```
def update_weights(network, row, l_rate):
    for i in range(len(network)):
        inputs = row[:-1]
        if i != 0:
            inputs = [neuron['output'] for neuron in network[i - 1]]
        for neuron in network[i]:
            for j in range(len(inputs)):
                neuron['weights'][j] += l_rate * neuron['delta'] * inputs[j]
            neuron['weights'][-1] += l_rate * neuron['delta']
```

-Train Network:
First loop is for a fixed number of epochs, and within each epoch updating the network for each row in the training data set.

```
def train_network(network, train, l_rate, n_epoch, n_outputs):
    for epoch in range(n_epoch):
        for row in train:
            outputs = forward_propagate(network, row)
            expected = [0 for i in range(n_outputs)]
            expected[row[-1]] = 1
            backprop_err(network, expected)
            update_weights(network, row, l_rate)
```

v. Prediction:
The implemented function returns the index in the network output that has the largest probability, assuming that the class values have been converted into integers starting at 0.

```
def predict(network, row):
    outputs = forward_propagate(network, row)
    return outputs.index(max(outputs))
```

vi. Results:
The implemented code consists of five neurons in the hidden layer and 3 neurons in the output layer. The network was trained for 500 epochs with a learning rate of 0.3.
Running the code will produce the classification with an accuracy of 93%.
A sample wheat seed data set was downloaded from: http://archive.ics.uci.edu/ml/datasets/seeds
The entire code can be found in:
https://github.com/fierro-cristina/Applied-Mathematics/blob/master/backprop-seed.py
Note: The algorithm uploaded can contain slight changes or updates. The sample database was also uploaded for further use commodities.

4) Conclusions:
The use of back propagation in the classification algorithm allowed for a faster, much more efficient classification of data, making the algorithm ideal in speed and size. Also, the implementation of forward propagation allowed for a more correct output (less error rate).