Neural Networks - Neural network training

Andrzej Kordecki

Neural Networks for Classification and Identification (ML.EM05): Exercise 09
Division of Theory of Machines and Robots
Institute of Aeronautics and Applied Mechanics
Faculty of Power and Aeronautical Engineering
Warsaw University of Technology

Table of Contents

Neural network model

Neural network model:

- Neural network will consist of many single neuron dictionaries. The dictionary structure will be based on previous exercises.
- Definition network will be based on layers of the neural network with use of many model of neurons. The function for weight generation will be included into network definition.
- The parameters of layers will be shared between neurons. Therefore there will be no difference in structure between neuron in the same layer.
- The presented model will be limited to the fully connected (dense) layers, i.e. to building the MLP network.
- The order of the layers will follow the order of dictionary structure.

Neural network model

The new neural network model dictionary:

```
structure = [
{'type': 'input', 'units': 1},
{'type': 'dense', 'units': 4, 'activation_function': 'I
                'bias': True},
{'type': 'dense', 'units': 4, 'activation_function': 'I
                'bias': True},
{'type': 'dense', 'units': 1, 'activation_function': 'I
                'bias': True}]
```

Neural network model

Neural network model function:

```
def create_network(self, structure):
    self.nnetwork = list()
    for index, layer in enumerate(structure[1:],
                                  start=1):
        new_layer = []
        for i in range(layer['units']):
            .
            (... code on the next slide ...)
            .
        self.nnetwork.append(new_layer)
    return self.nnetwork
```

Neural network model

In each layer we define each neuron:

```
neuron = {
    'weights': [np.random.randn() for i in
                range(structure[index - 1]['units']
                + int(layer['bias']))],
    'bias': layer['bias'],
    'activation_function': layer['activation_function']
    'activation_potential': 0,
    'delta': [0 for i in
                range(structure[index - 1]['units']
                + int(layer['bias']))],
    'output': 0}
new_layer.append(neuron)
```

Neural network model

Creating neural network with one input layer, 2 hidden layers, and one output.

```
structure = [
{'type': 'input', 'units': 1},
{'type': 'dense', 'units': 4,
                'activation_function': 'logistic',
                'bias': True},
{'type': 'dense', 'units': 4,
                'activation_function': 'logistic',
                'bias': True},
{'type': 'dense', 'units': 1,
                'activation_function': 'linear',
                'bias': True}]
model = Neural_network()
network = model.create_network(structure)
```

Neural network model

Creating neural network:

```
for layer in network:
    print(layer)
# Prints:
[{'weights': [-0.0152265760395027, -0.26997145675195 2],
'bias': True, 'activation_function': 'logistic',
'activation_potential': 0, 'delta': [0, 0],
'output': 0},
{'weights': [-0.474846771578107, 2.00405596548733],
'bias': True, 'activation_function': 'logistic',
'activation_potential': 0, 'delta': [0, 0],
'output': 0},
......
```

Back propagation algorithm

Back propagation algorithm changes:

1. In the forward phase - the change mainly relates to the possibility of transmitting a signal between the layers. We will use neuron characteristics that each neuron of the present layer is connected with reach neuron of the next layer.
2. In the backward phase - should take into account the propagation of the error signal using δ through whole network. It requires a separate δ calculation method to be defined for the last layer and another method for all other layers.

Back propagation algorithm

The changes in the forward phase:

```
def forward_propagate(self, nnetwork, inputs):
    row = list(inputs.copy())
    for layer in nnetwork:
        .
        (... code on the next slide ...)
        .
    return row
```

Back propagation algorithm

The changes in the forward phase:

```
next_row = []
for neuron in layer:
    bias_inputs = row.copy()
    if neuron['bias']:
        bias_inputs.append(1)
    tf = neuron_fcn()
    neuron['activation_potential'] =
                tf.activation_potential(neuron, row)
    neuron['output'] =
                tf.output(neuron, derivative=False)
    next_row.append(neuron['output'])
row = next_row.copy()
```

Back propagation algorithm

The δ information will contains network error as fallows:

- for output layer:

$$\delta_k^{(N)} = E'(y_j^{(N)}, t_j) f'(\sum_j y_j^{(N-1)} w_{j,k}^{(N-1)})$$

We have previously defined the derivative of loss function and weight update for the last layer of neural network. Now we need only to expand changes for all neurons of last layer.

Back propagation algorithm

The δ information will contains network error as fallows:

- for hidden layers:

$$\delta_k^{(n)} = (\textstyle\sum_k \delta_k^{(n+1)} w_{g,k}^{(n)}) f'(\sum_j y_j^{(n-1)} w_{j,k}^{(n-1)})$$

  We need to add this new calculation formula to the code. But, neuron structure is already prepared to store error information in neuron dictionary key "delta". Both of the function (for last and hidden layers) have the second element the same - derivative of activation function.

Weight update equation can be written as:

$$\Delta w_{h,g}^{(n)} = \eta \sum_p \delta_g^{(n+1)} y_h^{(n)}$$

Back propagation algorithm

Definition of back propagation function:

```
def backward_propagate(self, loss_function,
                       nnetwork, expected):
    for i in reversed(range(len(nnetwork))):
        layer = nnetwork[i]
        errors = list()
        if i != len(nnetwork) - 1:
            (... for all hidden layers ...)
            (... code on the next slide ...)
        else:
            (... for last layer ..)
        for j in range(len(layer)):
            (... delta value update ...)
```

Back propagation algorithm

Definition of back propagation function:

```
if i != len(nnetwork) - 1: # hidden layers
    for j in range(len(layer)):
        error = 0.0
        for neuron in nnetwork[i + 1]:
            error += (neuron['weights'][j]
                      * neuron['delta'])
        errors.append(error)
else: # last layer
    for j in range(len(layer)):
        neuron = layer[j]
        loss = loss_fcn()
        errors.append(loss.loss(loss_function,
                      expected[j], neuron['output'],
                      derivative=True))
```

Back propagation algorithm

Delta value update:

```
for j in range(len(layer)):
    tf = neuron_fcn()
    neuron = layer[j]
    neuron['delta'] = errors[j]
            * tf.output(neuron, derivative=True)
```

Weight update equation is performed on the all neurons in the same way:

$$\Delta w_{h,g}^{(n)} = \eta \sum_{p} \delta_g^{(n+1)} y_h^{(n)}$$

We need only add information from forward phase about neuron input values.

Back propagation algorithm

Weight update:

```
def update_weights(self, nnetwork, inputs, l_rate):
    for i in range(len(nnetwork)):
        row = inputs
        if i != 0:
            row = [neuron['output'] for
                        neuron in nnetwork[i - 1]]

        for neuron in nnetwork[i]:
            for j in range(len(row)):
                neuron['weights'][j] -= l_rate
                        * neuron['delta'] * row[j]
            if neuron['bias']:
                neuron['weights'][-1] -= l_rate
                        * neuron['delta']
```

New classes

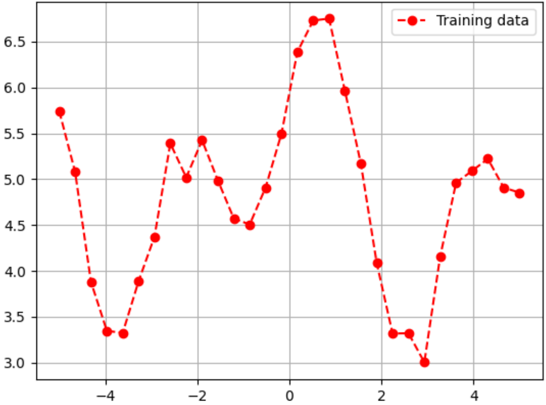
Training and predict function did not change from the previous exercises. The training can be initialized with:

```
# Create network
model = Neural_network()
structure = [{'type': 'input', 'units': 1},
             {'type': 'dense', 'units': 4,
                'activation_function': 'linear',
                'bias': True},
             {'type': 'dense', 'units': 1,
                'activation_function': 'linear',
                'bias': True}]

network = model.create_network(structure)
model.train(network, X, Y, 0.01, 4000, 'mse')
```

Regression example

Regression data:



Regression example

Data generation is based on sine and cosine functions:

```
# Generate regression dataset
X = np.linspace(-5, 5, n).reshape(-1, 1)
y = np.sin(2 * X) + np.cos(X) + 5
# simulate noise
data_noise =
    np.random.normal(0, 0.2, n).reshape(-1, 1)
# Generate training data
Y = y + data_noise
```

Regression example

Regression example 1 - optimal structure:

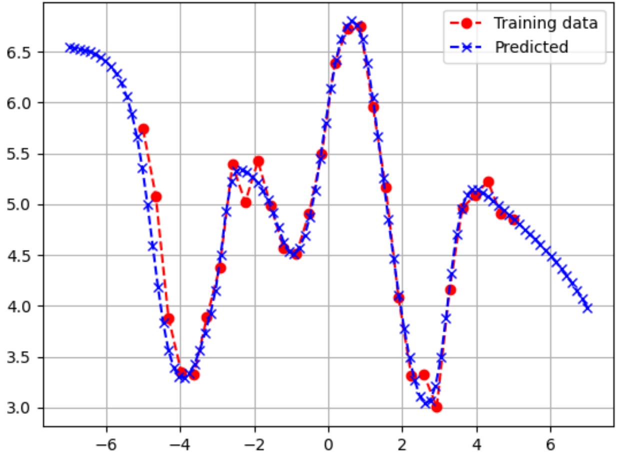
- Neural network structure:

```
{'type': 'input', 'units': 1},
{'type': 'dense', 'units': 8,
    'activation_function': 'tanh', 'bias': True},
{'type': 'dense', 'units': 8,
    'activation_function': 'tanh', 'bias': True},
{'type': 'dense', 'units': 1,
    'activation_function': 'linear', 'bias': True}
```

- Training parameters: binary cross entropy, learning rate: 0.01, epochs: 2000.
- Loss: 0.350.

Regression example

Regression example 1.



Regression example

Regression example 2 - too large structure:

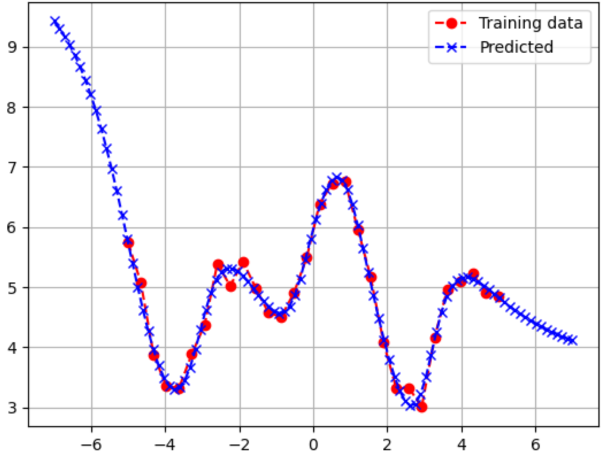
- Neural network structure:

```
{'type': 'input', 'units': 1},
{'type': 'dense', 'units': 16,
    'activation_function': 'tanh', 'bias': True},
{'type': 'dense', 'units': 16,
    'activation_function': 'tanh', 'bias': True},
{'type': 'dense', 'units': 1,
    'activation_function': 'linear', 'bias': True}
```

- Training parameters: binary cross entropy, learning rate: 0.01, epochs: 8000.
- Loss: 0.505.

Regression example

Regression example 2.



Regression example

Regression example 3 - too small structure:

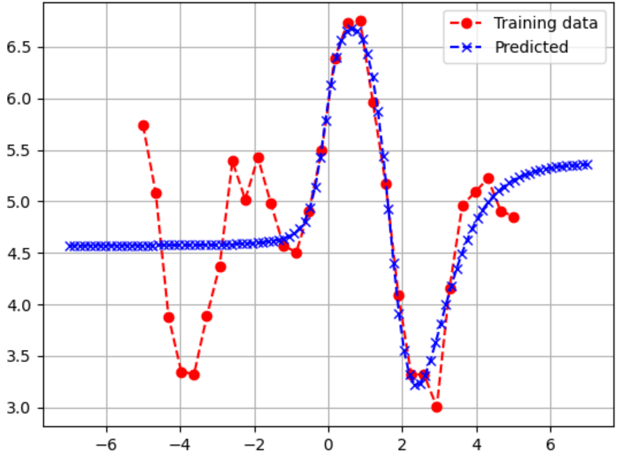
- Neural network structure:

```
{'type': 'input', 'units': 1},
{'type': 'dense', 'units': 4,
    'activation_function': 'tanh', 'bias': True},
{'type': 'dense', 'units': 4,
    'activation_function': 'tanh', 'bias': True},
{'type': 'dense', 'units': 1,
    'activation_function': 'linear', 'bias': True}
```

- Training parameters: binary cross entropy, learning rate: 0.01, epochs: 4000.
- Loss: 4.441.

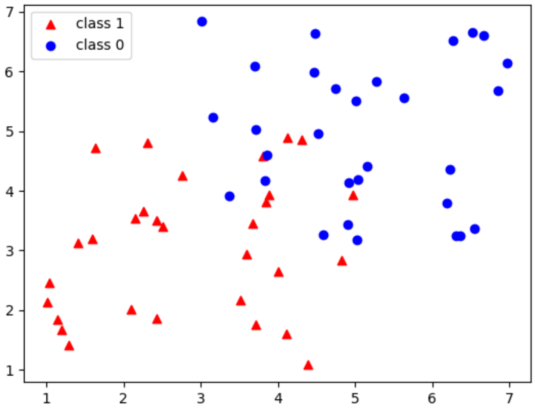
Regression example

Regression example 3.



Classification example

Classification data:



Classification example

Data generation to classes near to distinct points:

```
# Class 1 - samples generation
X1_1 = 1 + 4 * np.random.rand(n, 1)
X1_2 = 1 + 4 * np.random.rand(n, 1)
class1 = np.concatenate((X1_1, X1_2), axis=1)
Y1 = np.ones(n)

# Class 0 - samples generation
X0_1 = 3 + 4 * np.random.rand(n, 1)
X0_2 = 3 + 4 * np.random.rand(n, 1)
class0 = np.concatenate((X0_1, X0_2), axis=1)
Y0 = np.zeros(n)

X = np.concatenate((class1, class0))
Y = np.concatenate((Y1, Y0))
```

Classification example

Classification example 1 - line borders:

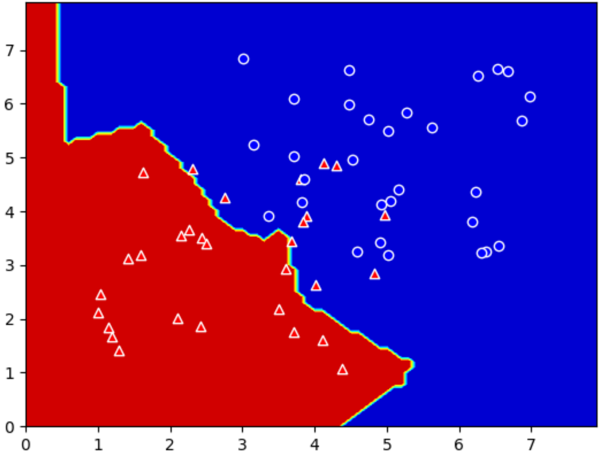
- Neural network structure:

```
{'type': 'input', 'units': 1},
{'type': 'dense', 'units': 8,
    'activation_function': 'relu', 'bias': True},
{'type': 'dense', 'units': 8,
    'activation_function': 'relu', 'bias': True},
{'type': 'dense', 'units': 1,
    'activation_function': 'logistic',
    'bias': True}
```

- Training parameters: binary cross entropy, learning rate: 0.01, epochs: 1000.
- Loss: 15.059 (erratic changes), accuracy: 82%

Classification example

Classification example 1.



Classification example

Classification example 2 - round borders:

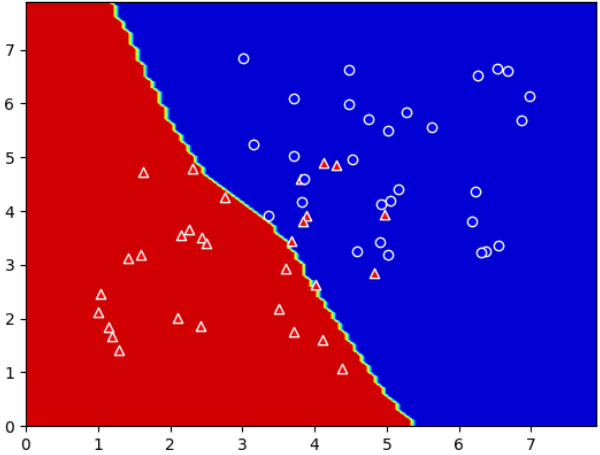
- Neural network structure:

```
{'type': 'input', 'units': 1},
{'type': 'dense', 'units': 8,
     'activation_function': 'logistic',
     'bias': True},
{'type': 'dense', 'units': 8,
     'activation_function': 'logistic',
     'bias': True},
{'type': 'dense', 'units': 1,
     'activation_function': 'logistic',
     'bias': True}
```

- Training parameters: binary cross entropy, learning rate: 0.01, epochs: 1000.
- Loss: 14.565, accuracy: 87%

Classification example

Classification example 2.



Classification example

Classification example 3 - to simple structure?:

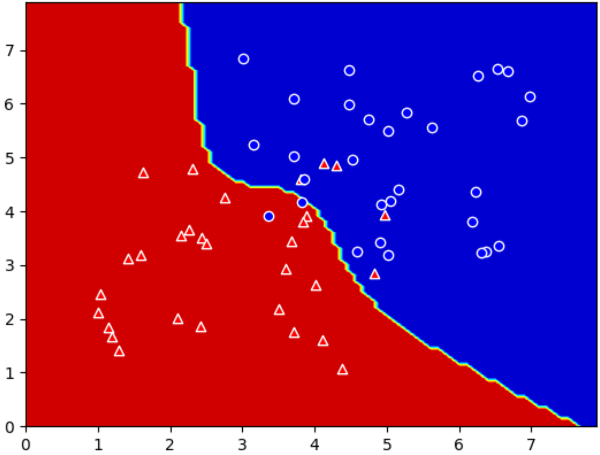
- Neural network structure:

```
{'type': 'input', 'units': 1},
{'type': 'dense', 'units': 8,
     'activation_function': 'logistic',
     'bias': True},
{'type': 'dense', 'units': 8,
     'activation_function': 'logistic',
     'bias': True},
{'type': 'dense', 'units': 1,
     'activation_function': 'logistic',
     'bias': True}
```

- Training parameters: binary cross entropy, learning rate: 0.002, epochs: 5000.
- Loss1: 15.559, acc1: 87%, Loss2: 15.442, acc2: 87%

Classification example

Classification example 3.



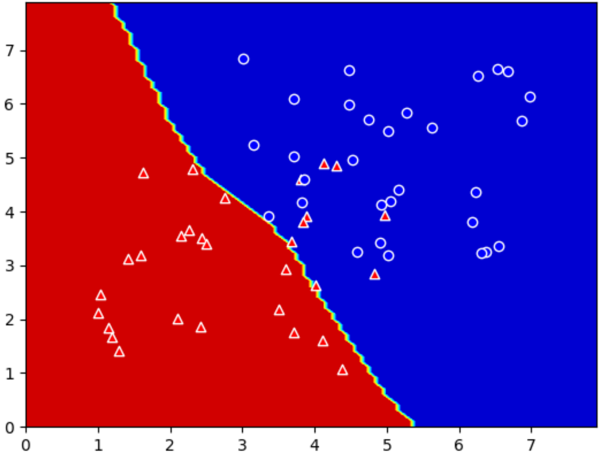
Classification example

Classification example 1 - to short calculation time?:

- Neural network structure:
```
{'type': 'input', 'units': 1},
{'type': 'dense', 'units': 24,
    'activation_function': 'logistic',
    'bias': True},
{'type': 'dense', 'units': 12,
    'activation_function': 'logistic',
    'bias': True},
{'type': 'dense', 'units': 1,
    'activation_function': 'logistic',
    'bias': True}
```
- Training parameters: binary cross entropy, learning rate: 0.01, epochs: 1000.
- Loss: 14.565, accuracy: 85%

Classification example

Classification example 4.



Classification example

Classification example 4:

- Neural network structure:

```
{'type': 'input', 'units': 1},
{'type': 'dense', 'units': 24,
    'activation_function': 'logistic',
    'bias': True},
{'type': 'dense', 'units': 12,
    'activation_function': 'logistic',
    'bias': True},
{'type': 'dense', 'units': 1,
    'activation_function': 'logistic',
    'bias': True}
```

- Training parameters: binary cross entropy, learning rate: 0.002, epochs: 5000.
- Loss: 13.961, accuracy: 90%

Classification example

Classification example 4.