Neural Networks - Neuron model and loss functions in Python

Andrzej Kordecki

Neural Networks for Classification and Identification (ML.EM05): Exercise 07
Division of Theory of Machines and Robots
Institute of Aeronautics and Applied Mechanics
Faculty of Power and Aeronautical Engineering
Warsaw University of Technology

Python code

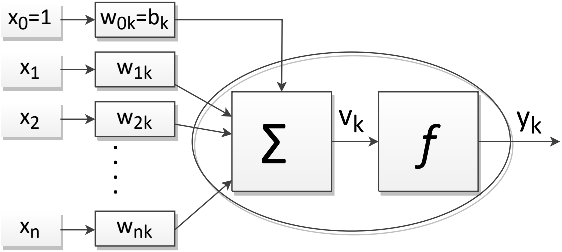
Notice:

The code shown during the exercises is not intended to be the best optimized solution or the shortest implementation. The code is designed to best illustrate the processes and data flow of neural networks.

Artificial Neuron

Neuron model

Neuron model:



Input:

$u_k = \sum_{i=1}^{n} w_{ik} x_i$

$v_k = \sum_{i=1}^{n} w_{ik} x_i + w_{0k}$

Output:

$y_k = f(v_k) = f(x_i, w_{ik})$

Neuron model

Neuron should be defined as class or dictionary, with include information about:

- w_{ik} - weights,
- $w_{0k} = b_k$ - bias,
- v_k - effective input of neuron (local field or activation potential),
- f - activation function,
- y_k - output of neuron k

Additionally we must now information about input data.

Neuron model

Neuron model definition:

```
n = 5 # number of inputs
input = np.random.rand(n, 1)

neuron = {"weights": None,
          "bias": True,
          "activation_potential": None,
          "activation_function": "sigmoid",
          "output": None
          }
print(neuron) # Print
{'weights': None, 'bias': True, 'activation_potential':
'activation_function': 'sigmoid', 'output': None}
```

Neuron model

Initial weights can be generate with random function from Numpy library:

- numpy.random.seed(seed=None) - seed the generator, which makes the random numbers more predictable.
- numpy.random.randn(d0, d1, ..., dn) - return a random values from the "normal" (Gaussian) distribution of mean 0 and variance 1.
- numpy.random.rand(d0, d1, ..., dn) - return a random values from a uniform distribution over [0, 1).

Neuron model

Example:

```
import numpy as np

np.random.seed(4)
print(np.random.rand(4))
print(np.random.rand(4))

# Prints every time we run the program:
[0.96702984 0.54723225 0.97268436 0.71481599]
[0.69772882 0.2160895  0.97627445 0.00623026]
```

Neuron model

Weights generation:

```
def generate_weights(neuron, number):
    neuron['weights'] = [np.random.randn()
                         for i in range(number
                         + int(neuron['bias']))]
    return neuron

# Print
{'weights': [0.22733602246716966, 0.31675833970975287,
0.7973654573327341, 0.6762546707509746,
0.391109550601909, 0.33281392786638453],
'bias': True, 'activation_potential': None,
'activation_function': 'sigmoid', 'output': None}
```

Neuron model

Activation potential (local field):

$$v_k = \sum_{i=1}^{n} w_{ik} x_i + w_{0k}$$

Code:

```
def neuron_activation_potential(neuron, inputs):
    activation = 0
    if neuron["bias"]:
        inputs = np.append(inputs, 1)
    for i, weight in enumerate(neuron["weights"]):
        activation += weight * inputs[i]
    return activation
```

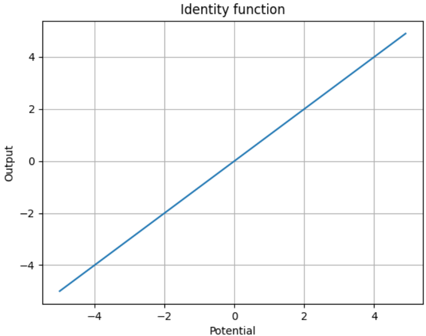
Identity function

Identity function:

$$y(v) = v$$

Code:

```
def neuron_linear(neuron):
    return neuron['activation_potential']
```



Hyperbolic Tangent function

Hyperbolic Tangent function:

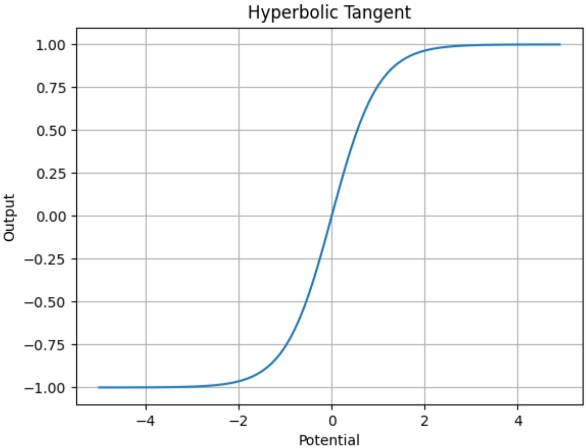
- Function:
  $$tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$$
- Output:
  $y(v) = tanh(v)$

Code:

```
def neuron_tanh(neuron):
    return (np.exp(neuron['activation_potential'])
            - np.exp(-neuron['activation_potential']))
            / (np.exp(neuron['activation_potential'])
            + np.exp(-neuron['activation_potential']))
```

Hyperbolic Tangent function

Hyperbolic Tangent function:



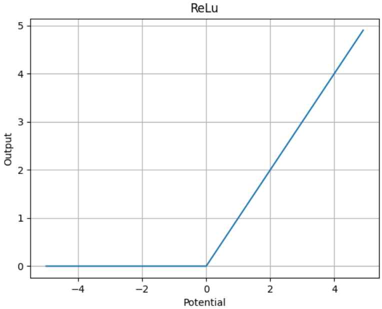
ReLu function

Rectified linear unit (ReLU)

$$y = \max(0, v)$$

Code:

```
def neuron_relu(neuron):
    return np.maximum(0, neuron['activation_potential'
```



Neuron

Example of neuron output calculation:

```
n = 5 % number of input values

input = np.random.rand(n, 1)

neuron = {"weights": None,
          "bias": True,
          "activation_potential": None,
          "activation_function": "sigmoid",
          "output": None
          }
```

Neuron

Example of neuron output calculation:

```
generate_weights(neuron, n)
neuron["activation_potential"] =
        neuron_activation_potential(neuron, input)
neuron["output"] = neuron_tanh(neuron, False)

print(neuron) # Prints
# {'weights': [-0.14623749504660263, 1.044761128213961,
0.6265018140844549, 0.379650100589177,
-0.005760602815801898, -0.8629530047653525],
'bias': True,
'activation_potential': -0.03153907988342153,
'activation_function': 'sigmoid',
'output': -0.031528626592471735}
```

Loss function

The loss function of one neuron used to adjust the network weights w:

$$E(w) = \sum_p f(t(p), y(x(p), w))$$

Where:

- p - is the sample,
- $t(p)$ - is desired output value of neuron and p training sample,
- y - is predicted by neural network output value for input $x(p)$ training sample.

The loss function is summation over samples and output neurons.

Loss function

The function:

```
def loss_fcn(loss, expected, outputs):
    loss = str.lower(loss) # convert to lower case
    error_sum = 0
    if loss == 'mse':
        error_sum = mse(expected,
                        outputs)
    elif loss == "binary_cross_entropy":
        error_sum = binary_cross_entropy(expected,
                                         outputs)
    return error_sum
```

Loss function

Mean Square Error (MSE, L2 Loss):

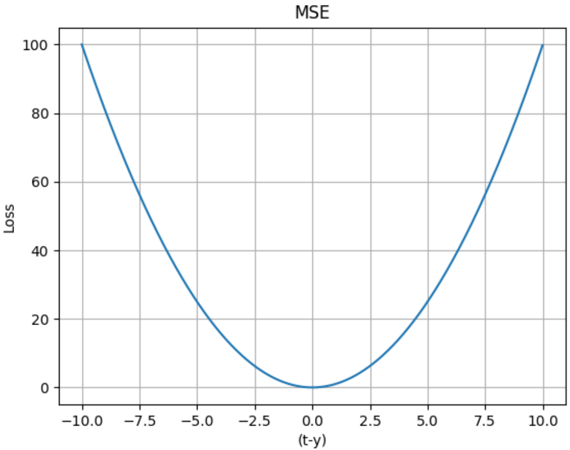
$$L = \frac{1}{N}\sum_{i=1}^{N}(y_i - \hat{y}_i)^2$$

Code:

```
def mse(expected, outputs):
    return error_value = (expected - outputs) ** 2
```

Loss function

Mean Square Error in range $< -10, 10)$:



Loss function

Binary cross entropy loss function:

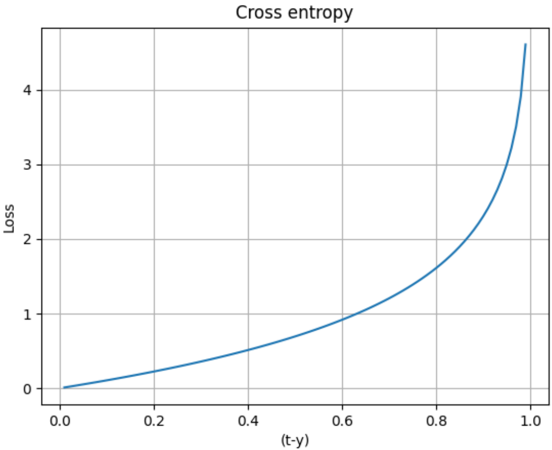
$$L = -\frac{1}{N}\sum_{k=1}^{N}\sum_{i=1}^{C} t_{k,i} \log(f(x_{k,i}))$$

Code:

```
def binary_cross_entropy(expected, outputs):
    return error_value = -expected * np.log(outputs)
                + (1-expected) * np.log(outputs)
```

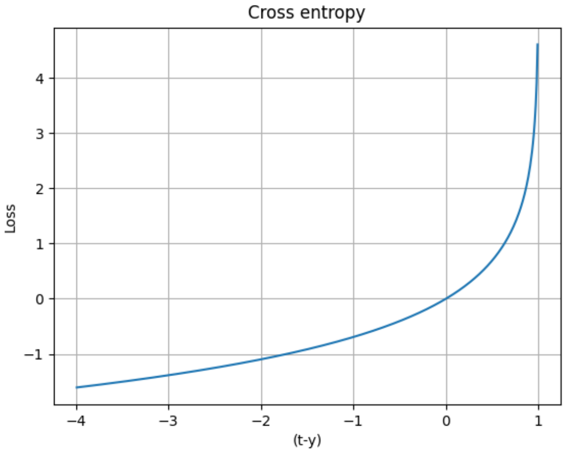
Loss function

Binary cross entropy loss function in range $(0, 1 >$:



Loss function

Binary cross entropy loss function in range $(-4, 4>$:



Loss function

The zip() function takes several iterables in parallel, aggregates them in a tuple with an item from each one, and returns it.

```
for item in zip([1, 2, 3], ['sugar', 'spice',
                             'everything nice']):
    print(item)

# Prints:
# (1, 'sugar')
# (2, 'spice')
# (3, 'everything nice')
```

Loss function

Example:

```
t = np.arange(-10, 10, 1)
yout = np.zeros((np.size(t)))

L = []
for (x, y) in zip(t, yout):
    E = loss_fcn("MSE", x, y)
    print(E)
    L.append(E)
print(f"L = {L}")

# Prints:
L = [100.0, 81.0, 64.0, 49.0, 36.0, 25.0, 16.0,
9.0, 4.0, 1.0, 0.0, 1.0, 4.0, 9.0, 16.0, 25.0,
36.0, 49.0, 64.0, 81.0]
```

Loss function

Example:

```
yout = np.arange(0, 1, 0.1)
t = np.ones((np.size(yout)))

L = []
for (x, y) in zip(t, yout):
    E = loss_fcn("binary_cross_entropy", x, y)
    print(E)
    L.append(E)
print(f"L = {L}")

# Prints:
L = [nan, 2.3025850929940455, 1.6094379124341003, 1.20
0.916290731874155, 0.6931471805599453, 0.510825623765
0.35667494393873, 0.2231435513142097, 0.1053605156578
```

Back-Propagation algorithm

The training method will use Back-Propagation algorithm consists of two phases:

1. In the forward phase - synaptic weights of the network are fixed and the function signal is propagated through the network from input, layer by layer, until it reaches the output.
2. In the backward phase - error signal is produced by comparing the output of the network with a desired response. In this second phase, successive adjustments are made to the synaptic weights of the network.

Derivatives

We want to adjust the network weights $w_{ij}^{(n)}$ in order to minimize loss function E by weight updates in direction opposite to gradient:

$$w_{h,g}^{new} = w_{h,g}^{old} - \eta \frac{\partial E}{\partial w_{h,g}}\Big|_{w_{h,g}=w_{h,g}^{old}}$$

Therefore we need to calculate the following values:

$$\frac{dE}{dw} = \frac{dE}{dy}\frac{dv}{dw}$$

We must know the derivatives of activation function $\frac{dy}{dv}$ and loss function $\frac{dE}{dy}$.

Identity function

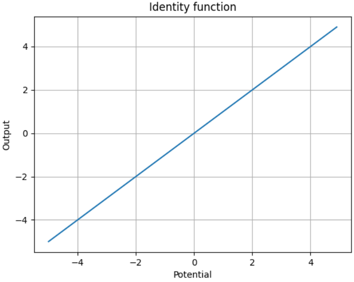
We can modify previous definition of identity function:

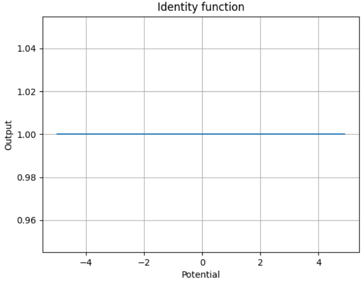
$$y(v) = v$$

Code:

```
def neuron_linear(neuron, derivative):
    out = 0
    if not derivative:
        out = neuron['activation_potential']
    else:
        out = 1
    return out
```

Identity function





Hyperbolic Tangent function

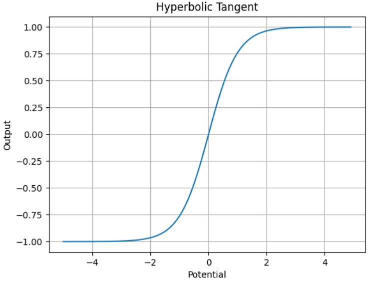
Modification of hyperbolic Tangent function:

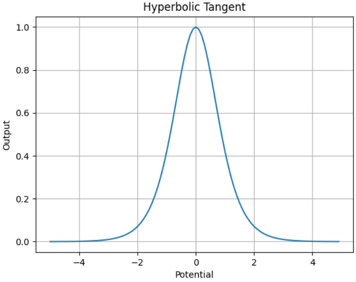
$$y(v) = tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$$

Code:

```
def neuron_tanh(neuron, derivative):
    out = 0
    if not derivative:
        out = (np.exp(neuron['activation_potential'])
            - np.exp(-neuron['activation_potential
            / (np.exp(neuron['activation_potential
            + np.exp(-neuron['activation_potential
    else:
        out = 1.0 - np.power(neuron['output'], 2)
    return out
```

Hyperbolic Tangent function





ReLu function

Rectified linear unit (ReLU)

$$y = \max(0, v)$$

Code:

```
def neuron_relu(neuron, derivative):
    out = 0
    if not derivative:
        out = np.maximum(0, neuron['activation_potentia
    else:
        if neuron['activation_potential'] >= 0:
            out = 1
    return out
```

ReLu function