# BackPropagation

In order to understand backpropagation, you will implement a backpropagation algorithm for a neural network from scratch. Backpropagation can be used for both classification and regression problems, we will focus on classification in this exercise.

**Exe. 1** Write a function namely `initialize_network(n_inputs, n_hidden, n_outputs)` that creates a neural network with an input, a hidden layer and an output layer.The parameters are the number of inputs, the number of neurons to have in the hidden layer and the number of outputs respectively.
Each layer of hidden or output layer contains `n_hidden, n_outputs` nodes respectively. For each layer, each node has a **weight**, plus an additional weight (constant) for the bias such as: $w_1x_1 + \cdots w_nx_n + w_0$.

```python
# Initialize a network
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 should be defined here
  #output layer should be added to the network here
  return network
```

for the hidden layer we create `n_hidden` neurons and each neuron in the hidden layer has `n_inputs` +1 weights, one for each input column in a dataset and an additional one for the bias. Layers as arrays of dictionaries and the whole network is an array of layers. The network weights are initialize to small random numbers in the range of 0 to 1. Complete the two missing line of code by yourself.

**Exe. 2** Now test your function by initializing a small network and printing each layer. Notice that since you use a random generator, you need to define a seed. You can define it by `seed(number)`. This number can be 1 or your student number.

**Exe. 3** We can calculate an output from a neural network by propagating an input signal through each layer until the output layer outputs its values. The first step is to calculate the activation of one neuron given an input. Neuron activation is calculated as the weighted sum of the inputs. Much like linear regression:

```
activation = sum(weight_i * input_i) + bias                    (1)
```

Where weight is a network weight, input is an input[1], i is the index of a weight or an input and bias is a special weight that has no input to multiply with (or

[1]It can be a line of the dataset and it should be a vector of the same size as the number of nodes in your defined network.

you can think of the input as always being 1.0).
Write a function namely `activate(weights, inputs)` with two parameters `weights` as a list of given weights and `inputs` as a list of inputs. For the sake of simplicity assume that the biased is the last weight in the list of weights. This function returns the activation value of the above formula (1).
**Exe. 4** Once a neuron is activated, we need to transfer the activation to see what the neuron output actually is. Different transfer functions can be used such as sigmoid activation function, tanh (hyperbolic tangent) function or rectifier transfer[2] function. choose one of this function as your transfer function and implement it as a function. For instance for the sigmoid function we should compute:

```
output = 1 / (1 + e^(-activation))                              (2)
```

Write a function `transfer(activation)` that receives the activation value and returns back the output using your transfer function.
**Exe. 5 Forward Propagation** Now we need to work through each layer of our network to calculate the outputs of each neuron. All of the outputs from one layer become inputs to the neurons on the next layer. Write a function name `forward_propagate(network, row)` that implements the forward propagation for a row of data from our dataset with our neural network

```
# Forward propagate input to a network output
def forward_propagate(network, row):
  inputs = row
  for layer in network:
    new_inputs = []
    for neuron in layer:
      neuron['output'] = #implement activate and transfer functions
     on the neuron weight and the given input row data. And assign
    the returned value to neuron['output']
      new_inputs.append(neuron['output'])
    inputs = new_inputs
  return inputs
```

A neuron's output value is stored in the neuron with the name 'output'. We collect the outputs for a layer in an array named `new_inputs` that becomes the array inputs and is used as inputs for the following layer. The function returns the outputs from the last layer (output layer). Notice that in this code the transfer function is added for both hidden and output layers. For the `row` parameter you can define it by yourself. It should be a selected vectors of the same size as `n_inputs` for instance `inputs = [random.randint(0, 1) for i in range(n_inputs)]`
**Exe. 6 (Test forward Propagation)** Test the `forward_propagate` function by defining a small network and a given row list example as an input. After implementing this function print the output value. Notice that the row size should be the same as the input size of your network and the final output should be the same size of your network output. For instance if you define a network as `network = initialize_network`$(3, 2, 2)$, the input row example should be a

[2]`https://en.wikipedia.org/wiki/Rectifier_(neural_networks)`

vector of size 3.
**Exe. 7** The back propagation error is calculated between the expected outputs and the outputs forward propagated from the network. These errors are then propagated backward through the network from the output layer to the hidden layer, assigning blame for the error and updating weights as they go.
**Transfer Derivative** First write a function to calculate the derivative of your transfer function w.r.t the output of your neural network `transfer_derivative(output)`. This is a calculus question. For instance if you choose the sigmoid as your transfer function its derivative should be calculated from the following formula: derivative = output * (1.0 - output)
**Exe 8 Error Backpropagation** The first step is to calculate the error for each output neuron, this will give us our error signal (input) to propagate backwards through the network. The error for a given neuron can be calculated as follows:
error = (expected - output) * transfer_derivative(output) (3)
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, as shown above.
This error calculation is used for neurons in the output layer. In the hidden layer, things are a little more complicated. The error signal for a neuron in the hidden layer is calculated as the weighted error of each neuron in the output layer. Think of the error traveling back along the weights of the output layer to the neurons in the hidden layer. The back-propagated error signal is accumulated and then used to determine the error for the neuron in the hidden layer, as follows:
error = (weight_k * error_j) * transfer_derivative(output) (4)
Where error_j is the error signal from the jth neuron in the output layer, weight_k is the weight that connects the kth neuron to the current neuron and output is the output for the current neuron. We want to write a function named `backward_propagate_error()` for implementing this procedure.

```python
# Backpropagate error and store in neurons
def backward_propagate_error(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 error signal calculated for each neuron is stored with the name 'delta'. The layers of the network should be iterated in reverse order, starting at the output and working backwards. This ensures that the neurons in the output layer have 'delta' values calculated first that neurons in the hidden layer can use in the subsequent iteration. The name 'delta' reflects the change the error implies on the neuron (e.g. the weight delta). Try to understand the code.

**Exe. 9** Test the `backward_propagate_error()` function with an output computed from exercise 6. Notice that the modified network from forward propagation now can be tested in this function with the same row input now as the predicted parameter. After executing the function, print your networks and layers of networks to see the results.

**Train Network** The network is trained using stochastic gradient descent (SGD). This involves multiple iterations of exposing a training dataset to the network and for each row of data forward propagating the inputs, backpropagating the error and updating the network weights. Let's break down the SGD into two sections: Update Weights and Train Network.