NEURAL NETWORKS – exercise 3

Exercise 3 covers the backpropagation algorithm, which you should already know from the lecture. The lecture provided formulas for individual parameters on various layers of neural networks and their derivation, and we will deal with implementation in the laboratories. Backpropagation is simply an application of the chaining rule:

$$\frac{\partial L}{\partial x} = \frac{\partial L}{\partial y}\frac{\partial y}{\partial x}$$

In calculating the model cost gradient by parameters. For the notation convention used:

- We operate on vectors and matrix multiplication, the order matters
- L is a scalar, in the notation where x is a column vector, $\frac{\partial L}{\partial x}$ is a vector with dimension corresponding to xT (so that matrix multiplication is correct)
- $\frac{\partial y}{\partial x}$ is the matrix of partial derivatives of all outputs over all inputs with dimensions ($dim_{y,\ dimx}$)

(There is also a reverse convention, if you come across it while searching for materials, here is a cheat sheet: https://en.wikipedia.org/wiki/Matrix_calculus#Layout_conventions)

How does this math translate into implementation? Let's take an example of any element-wise function, i.e. one that applies the transformation independently to each element of the vector, e.g.

$$y = f(x) = x2$$

For scalars, we know that the derivative of this function is f'(x) = 2x, so if we superimpose another function on it, so that h(x) = g(f(x)), then h'(x) = g'(x)2x.

For vectors we should have as $\frac{\partial y}{\partial x}$ a square matrix ($dim_{x,\ dimx}$) and use it in the chain rule, but note that all elements of this matrix except the diagonal will be zero, while the diagonal is simply the elements of the 2x vector.

Let's also remember that calculating the appropriate output values from each operation must be done in the order of these operations, while calculating the gradient after specific inputs - in the reverse order. Hence, if we want to implement the operation: squaring a vector, in practice we will do it something like this:

```
// x.shape=(x,1) to follow the convention described above
// write. Transposition will be needed in backward()def
forward(x):
     cache_x = x
     return x*x
```

```
//
derivative_y.shape=(1,x)def
backward(derivative_y):
     return derivative_y*2*cache_x.transpose()
```

What is important here is that:

- When moving forward, we need to save the value of the x vector for later forwarding after backward operations
- We can implement derivative/gradient as backward functions, which at the input it takes $\frac{\partial L}{\partial y}$ calculated so far, and at the output it gives $\frac{\partial L}{\partial x}$
- backward does not always have to be a full multiplication by $\frac{\partial y}{\partial x}$, e.g. above multiplication by a diagonal matrix is replaced by the numpy operator *, i.e. element-wise multiplication of vectors. The effect is the same we can benefit from simplification
- Within backward, if we use the learned model parameters, we should calculate and save the gradient - partial derivatives with respect to these parameters
- If every operation we use has these two functions implemented, we can easily build any sequence of them and calculate gradients at any level of this sequence

The task for the next two laboratories is to build a multi-layer neural network model with any activation function and cost function such as for logistic regression. Then you need to examine how the model behaves on the heart disease set for:

- Different dimensions of the hidden layer
- Different learning coefficient values
- Different standard deviations when initializing the weights
- Normalized and non-normalized data
- Different number of layers

The exercise is graded on a scale of 0-20 points.