Machine Learning for Communications
Neural Networks
Programming Exercises

Marcin Pikus

November 9, 2017

Institute for Communications Engineering
Technical University of Munich
Prof. Dr. sc. techn. Gerhard Kramer

The following lecture notes are part of the course "Machine Learning for Communications" offered by the Institute for Communications Engineering at the Technical University of Munich. All content is subject to copyright restrictions. If you are planning to use any of the material, please contact Prof. Dr. sc. techn. Gerhard Kramer (gerhard.kramer@tum.de).

1 Exercises — Set 3

1.1 Exercises

Recall the definition of Jacobian. Consider a function $f : \mathcal{R}^N \to \mathcal{R}^M$ and the vectors $\underline{x} \in \mathcal{R}^N$ and $\underline{y} = f(\underline{x}) \in \mathcal{R}^M$. We write the Jacobian as

$$\frac{\partial \underline{y}}{\partial \underline{x}} = \begin{bmatrix} \frac{\partial y_1}{\partial x_1} & \cdots & \frac{\partial y_1}{\partial x_N} \\ \vdots & \ddots & \vdots \\ \frac{\partial y_M}{\partial x_1} & \cdots & \frac{\partial y_M}{\partial x_N} \end{bmatrix}.$$

- Assume $\underline{z} = \mathbf{W}\underline{a} + \underline{b}$. Show that
$$\frac{\partial \underline{z}}{\partial \underline{b}} = \mathbb{I},$$
where $\mathbb{I}$ denotes the identity matrix.

- Assume $\underline{z} = \mathbf{W}\underline{a} + \underline{b}$. Show that
$$\frac{\partial \underline{z}}{\partial \underline{a}} = \mathbf{W}.$$

* Recall the Jacobian chain rule. Consider the functions $f\colon \mathcal{R}^N \to \mathcal{R}^M$, $g\colon \mathcal{R}^M \to \mathcal{R}^K$ and the vectors $\underline{x} \in \mathcal{R}^N$, $\underline{y} = f(\underline{x}) \in \mathcal{R}^M$, and $\underline{z} = g(\underline{y}) \in \mathcal{R}^K$. The Jacobian chain rule reads
$$\frac{\partial \underline{z}}{\partial \underline{x}} = \frac{\partial \underline{z}}{\partial \underline{y}} \frac{\partial \underline{y}}{\partial \underline{x}}.$$
Prove the Jacobian chain rule. Hint: consider a single entry in the matrix on the left hand side, i.e., $\frac{\partial z_i}{\partial x_j}$, and show that this entry is equal to an inner product of the i-th row from $\frac{\partial \underline{z}}{\partial \underline{y}}$ and the j-th column from $\frac{\partial \underline{y}}{\partial \underline{x}}$. The following identity (the total derivative) might be useful: assume a function of n variables $f = f(x_1, \ldots, x_n)$ where $x_i = x_i(t)$, then
$$\frac{\partial f}{\partial t} = \sum_{i=1}^{n} \frac{\partial f}{\partial x_i} \frac{\partial x_i}{\partial t}.$$

1.2 Programming exercises

In this exercises we will develop a class implementing a neural network. The class can be found in the file `mlcomm.nn.nnet`. Additionally, a test case was provided which can be used to test if the functions inside the `mlcomm.nn.nnet` were correctly implemented. For testing the implementation run the file `test_nn.py`. Of course, your implementation may be different (it will fail the test case) and still work correctly. However, if you follow the implementation guidelines below, your implementation should pass the taste case.

- Derivatives of the activation functions are extensively used in the backpropagation algorithm. Complete the function `mlcomm.nn.utils.dact_fct(x, type_fct)`. The function should behave analogously to `mlcomm.nn.utils.act_fct(x, type_fct)`, but should return the values of the derivatives of the cost functions. E.g., for the identity activation function it should return a vector or matrix of the same size as `x` but containing only ones.

1.2.1 Regression using NN

In this exercises we apply NNs to the regression problem. The main file: `nn_ex1.py`. In the regression problem we are going to use the quadratic cost function and the identity activation function for the output layer.

- Implement the function `init_Wb` in file `mlcomm.nn.nnet`. The function should initialize the matrices $\mathbf{W}^{[1]}, \ldots, \mathbf{W}^{[L]}$ and the vector $\underline{b}^{[1]}, \ldots, \underline{b}^{[L]}$ with random number from Gaussian distribution with 0 mean and 0.1 variance. You can run the test `test_nn.py` to check if the the sizes of matrices are correct.

- Implement the function `fp` in file `mlcomm.nn.nnet`. The function performs forward propagation. Recall the forward propagation-

$$\boxed{\begin{aligned} \underline{a}^{[0]} &= \underline{x} && (1)\\ \underline{z}^{[l]} &= \mathbf{W}^{[l]}\underline{a}^{[l-1]} + \underline{b}^{[l]} && (2)\\ \underline{a}^{[l]} &= g^{[l]}\left(\underline{z}^{[l]}\right) && (3) \end{aligned}}$$

  The input to the function is a matrix where each column contains a different input vector for NN, i.e,

$$\mathbf{X}_{\text{data}} = \left[\underline{x}^{(1)}, \quad \ldots, \quad \underline{x}^{(N)}\right]. \tag{4}$$

  The output consist of two dictionaries which contain the outputs and inputs for each layer (for all input vectors, each column corresponding to one input vector), i.e,

$$\begin{aligned} \texttt{a} = \{&\texttt{0}: \left[\underline{x}^{(1)}, \quad \ldots, \quad \underline{x}^{(N)}\right],\\ &\texttt{1}: \left[\underline{a}^{[1]}(\underline{x}^{(1)}), \quad \ldots, \quad \underline{a}^{[1]}(\underline{x}^{(N)})\right],\\ &\ldots\\ &\texttt{L}: \left[\underline{a}^{[L]}(\underline{x}^{(1)}), \quad \ldots, \quad \underline{a}^{[L]}(\underline{x}^{(N)})\right]\} \end{aligned}$$

$$\begin{aligned} \texttt{z} = \{&\texttt{1}: \left[\underline{z}^{[1]}(\underline{x}^{(1)}), \quad \ldots, \quad \underline{z}^{[1]}(\underline{x}^{(N)})\right],\\ &\texttt{2}: \left[\underline{z}^{[2]}(\underline{x}^{(1)}), \quad \ldots, \quad \underline{z}^{[2]}(\underline{x}^{(N)})\right],\\ &\ldots\\ &\texttt{L}: \left[\underline{z}^{[L]}(\underline{x}^{(1)}), \quad \ldots, \quad \underline{z}^{[L]}(\underline{x}^{(N)})\right]\} \end{aligned}$$

You can run the test `test_nn.py` to check if the forward propagation works properly.

- Implement the function `output` in file `mlcomm.nn.nnet`. The function returns the outputs of the NN for the input matrix containing the inputs to NN. The input matrix is organized as above. The output data should be organized in a matrix as above.

- Implement the function `bp` in file `mlcomm.nn.nnet`. The function performs backpropagation. Recall the backpropagation for the per-sample cost (for the general cost function).

$$\frac{\partial C^{(i)}}{\partial z^{[L]}} = \frac{\partial C^{(i)}}{\partial a^{[L]}} g'^{[L]}(z^{[L]}) \tag{5}$$

$$\frac{\partial C^{(i)}}{\partial \underline{z}^{[l]}} = \left(\frac{\partial C^{(i)}}{\partial \underline{z}^{[l+1]}} \mathbf{W}^{[l+1]} \right) \odot g'^{[l]}(\underline{z}^{[l]}) \tag{6}$$

$$\frac{\partial C^{(i)}}{\partial \underline{b}^{[l]}} = \frac{\partial C^{(i)}}{\partial \underline{z}^{[l]}} \tag{7}$$

$$\frac{\partial C^{(i)}}{\partial \mathbf{W}^{[l]}} = \left(\underline{a}^{[l-1]} \frac{\partial C^{(i)}}{\partial \underline{z}^{[l]}} \right)^T \tag{8}$$

The equation (5) contains the term $\frac{\partial C^{(i)}}{\partial a^{[L]}}$. This is the derivative of the cost function with respect to the output of the network. Notice that this is the only place where to cost function enters the backpropagation algorithm. This term is a function of the output of the network $a^{[L]}$ and the data y. We provide this function to the backpropagation algorithm via the function handle `dCda_func` as a parameter. This means that (5) could be implemented as:

`dCdz[L] = dCda_func(y,a[L])*dact_fct(z[L], self.act_out)`.

If the imput data are organized as in (4), this line will produce

$$\texttt{dCdz[L]} = \left[\frac{\partial C^{(1)}}{\partial z^{[L]}}, \quad \ldots, \quad \frac{\partial C^{(N)}}{\partial z^{[L]}} \right].$$

One iteration of the equation (6) could be implemented as

`dCdz[l] = self.W[l+1].T.dot(dCdz[l+1]) * dact_fct(z[l],self.act_hid)`,

which produces a matrix

$$\texttt{dCdz[l]} = \left[\left(\frac{\partial C^{(1)}}{\partial \underline{z}^{[l]}} \right)^T, \quad \ldots, \quad \left(\frac{\partial C^{(N)}}{\partial \underline{z}^{[l]}} \right)^T \right].$$

which stores the derivatives (6) in columns. Finally, the derivatives should be mapped to the derivatives (7) and (8) and summed over the samples to obtain the

total gradients which should be outputed by the backpropagation function. If you are having problems implementing the backpropagation you can take a look at the solution in the Appendix.

- Implement the function `gd_learn` in the file `mlcomm.nn.nnet`. The function should perform gradient descent with the gradients obtained form the backpropagation function. Update for $l = 1, \ldots, L$

$$\mathbf{W}^{[l]} \leftarrow \mathbf{W}^{[l]} - \beta \frac{\partial C}{\partial \mathbf{W}^{[l]}}$$
$$\underline{b}^{[l]} \leftarrow \underline{b}^{[l]} - \beta \left(\frac{\partial C}{\partial \underline{b}^{[l]}} \right)^T .$$

- Find the number of layers, the number of neurons, the activation functions, the number of GD iterations, and the learning rate to obtain low cost value. You may obtain a figure similar to Fig. 1 and the cost around 0.15.

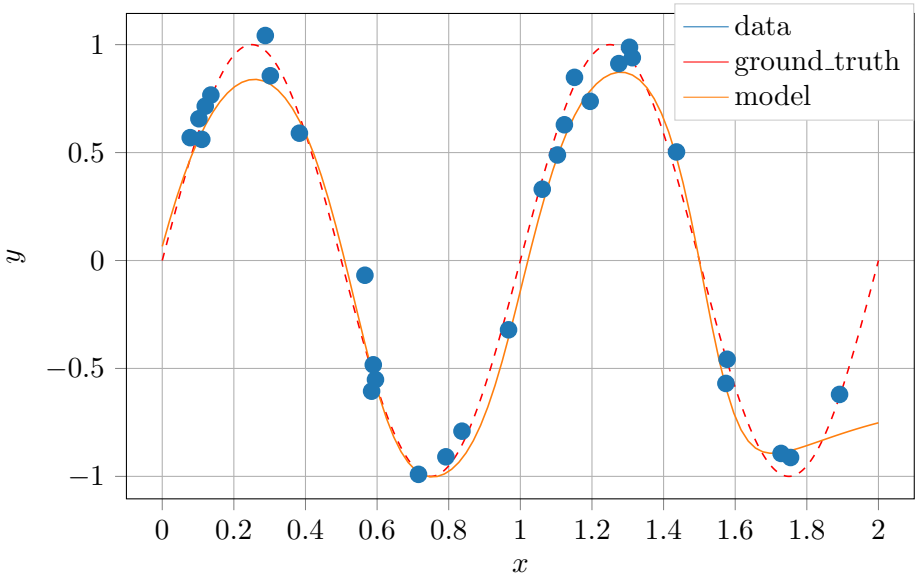

Figure 1: NN regression.

1.2.2 Binary classification using NN

In this exercises we apply NN to the binary classification (logistic regression) problem. The main file: `nn_ex2.py`. For the binary classification we are going to use the cross entropy cost function between the network output $a^{[L]}$ and the output variable from the training set y.

- Implement the function `cost` in the file `nn_ex2.py` which returns the cross entropy cost for the current output of the network.

$$C = \sum_{i=1}^{N} C^{(i)} = \sum_{i=1}^{N} -y^{(i)} \log(a^{(i)}) - (1 - y^{(i)}) \log(1 - a^{(i)})$$

- Implement the function `dCda_f` in the file `nn_ex2.py` which returns the derivative of the cost with respect to the network output (you need to derive the formula for the function). The function will be provided to the `gd_learn` function, which will in effect pass this function to the backpropagation algorithm. This function is essential for the first line (5) of the backpropagation and should compute the term $\frac{\partial C^{(i)}}{\partial a^{[L]}}$. Function should work on vectors, just as the same function for the regression problem.

- Find the number of layers, number of neurons, activation functions, number of GD iterations, and the learning rate to obtain low cost. You may obtain a figure similar to Fig. 2.

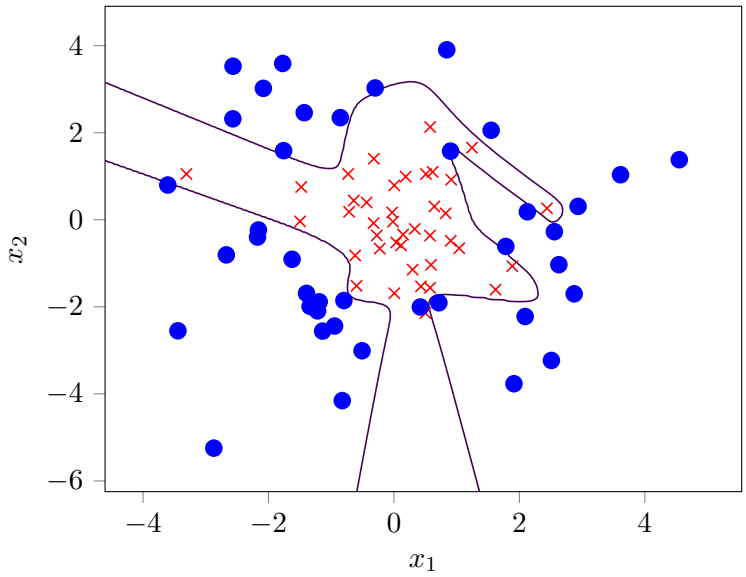

Figure 2: NN classification without regularization.

1.2.3 Binary classification using NN with regularization

We now add the regularization term to the cost function which results in the cost function

$$C = \sum_{i=1}^{N} C^{(i)} = \sum_{i=1}^{N} -y^{(i)} \log(a^{(i)}) - (1 - y^{(i)}) \log(1 - a^{(i)}) + \sum_{l=1}^{L} \lambda \|\mathbf{W}^{[l]}\|^2,$$

where $\|\mathbf{W}^{[l]}\|^2 = \mathrm{tr}\left(\mathbf{W}^{[l]}\left(\mathbf{W}^{[l]}\right)^T\right) = \sum_{i,j}\left(w_{i,j}^{[l]}\right)^2$. That is, the regularization corresponds to adding the sum of the squares of the entries from the matrices $\mathbf{W}^{[1]}, \ldots, \mathbf{W}^{[L]}$. Note that we do not normalize the bias terms.

- Derive the expression for the gradient (8) with for the cost function with the regularization. It could be done using the method form the lecture (differentiation with respect to rows of $\mathbf{W}^{[l]}$). Finally, we obtain

$$\frac{\partial C}{\partial \mathbf{W}^{[l]}} = \sum_{i=1}^{N} \frac{\partial C^{(i)}}{\partial \mathbf{W}^{[l]}} + \lambda \mathbf{W}^{[l]} = \sum_{i=1}^{N} \left(\underline{a}^{[l-1]} \frac{\partial C^{(i)}}{\partial \underline{z}^{[l]}} \right)^T + 2\lambda \mathbf{W}^{[l]}$$

- Modify the function `bp` with the new expression for the gradient. Modify the function `gd_learn` such that it calls the `bp` function with regularization parameter. Use $\lambda = 0.044$ to obtain result similar to the one on Fig. 3.

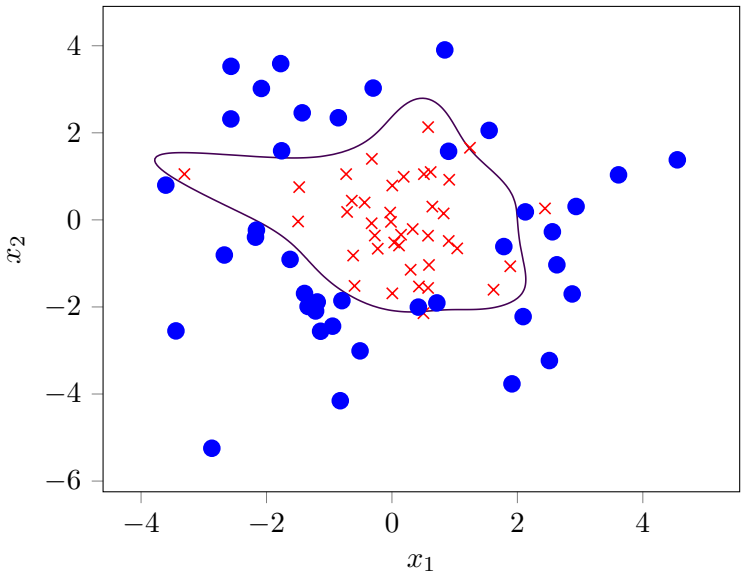

Figure 3: NN classification with regularization.

2 Appendix

Contains sample code for the backpropagation function.

```
def bp(self, y, x, dCda_func, lbd=0):
        a,z = self.fp(x)
        L = self.L

        dCdz = {L: dCda_func(y,a[L])*dact_fct(z[L], self.act_out) }
        for l in range(L-1,0,-1):
                dCdz[l] = self.W[l+1].T.dot(dCdz[l+1]) * dact_fct(z[l],self.act_hid)

        db = {}
        for l in range(1,L+1):
                db[l] = np.sum(dCdz[l], axis=1).reshape((-1,1))

        dW = {}
        for l in range(1,L+1):
                dW[l] = dCdz[l].dot(a[l-1].T)

        return dW, db
```