Machine Learning for Communications
Neural Networks
Programming Exercises

Marcin Pikus

October 25, 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 Programming Exercises — Set 1

This exercise set regards linear regression. We are given two one-dimensional data sets `dataset1_linreg`, `dataset2_linreg`. We are going to use the gradient descent (GD) algorithm during all exercises. For each of the exercises you may have to adjust the number of iterations and the step size of the GD to get a good performance. One technique to verify if the GD is behaving properly is to plot the cost versus the number of iterations. For plotting we will use `matplotlib` library (and Python3). You may install the library by `sudo apt-get install python3-matplotlib` (root password on the mlcomm vitrual machines is `mlcomm`). The files for the exercise can by executed by typing `python3 file_name.py` in the terminal. During implementation and debugging it is helpful to carefully observe the dimensions of the objects which enter/leave the calculations.

For code brevity, we are going to use slightly different notation than during the lecture. Instead of the separate bias term b in the linear regression, i.e.,

$$z = \sum_{k=1}^{M} w_k x_k + b \tag{1}$$

$$= \underline{w}^T \underline{x} + b, \tag{2}$$

we are going to replace b with w_0 and store it inside the parameters vector $\underline{w}$. We also need to extend the data vector $\underline{x}$ by additional 1. That is, we use the following notation

$$z = \sum_{k=1}^{M} w_k x_k + w_0 \tag{3}$$

$$= \begin{bmatrix} w_0 w_1 \dots w_M \end{bmatrix} \begin{bmatrix} 1 \\ x_1 \\ \vdots \\ x_M \end{bmatrix} \tag{4}$$

$$= \underline{w}^T \underline{x}. \tag{5}$$

All the derivations from the lecture should be adapted to this notation (exercise). The extension of the data set with additional 1 will be performed for you in the first exercise by the function `mlcomm.nn.utils.poly_extend_data1D(x)`. The function will take a vector of data points (for us $M = 1$ and a vector is used to store the input data in one place) and extend it with additional vector of $1s$. That is, if the input is

$$\begin{bmatrix} x^{(1)} \dots x^{(N)} \end{bmatrix}, \tag{6}$$

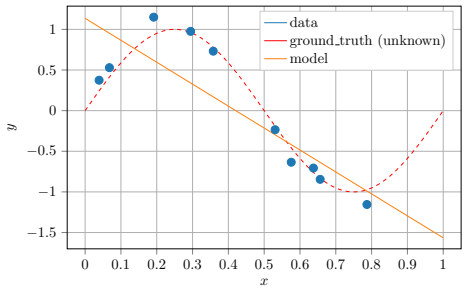
the output should be a matrix

$$\mathbf{X}_{\text{data}} = \begin{bmatrix} 1 & \dots & 1 \\ x^{(1)} & \dots & x^{(N)} \end{bmatrix} \tag{7}$$

1.1 Linear Regression

Complete the code inside the file: `lin_reg_ex1.py`.

- Complete the initialization of w after the line `#random init of w`. Initialize w with samples from the normal distribution.
- Complete the function `mlcomm.nn.utils.lir_cost(w, y, x)`, which computes the quadratic cost for the current set of parameters $\underline{w}$ and the training data. Your code should be *vectorized*, i.e., use rather matrix multiplications than `for` loops. This task can be computed in 1–2 lines when properly vectorized.
- Complete the function `mlcomm.nn.utils.lir_grad(w, y, x)`, which computes the gradient with respect to w for linear regression.
- Complete the function `mlcomm.nn.utils.gradient_descent(iter_num, l_rate, w_0, gradient_func)`. The last parameter is a function handle. This should be a function of `w` only, which returns the gradient for the given value of `w`. The gradient also depends on the training data, however it does not make sense for the gradient decent algorithm to know the data. Therefore we have created a wrapper function `gradient_wrapper` which computes the gradient and hides the data from the gradient decent algorithm. The handle to the wrapper function is provided to the gradient descent function.
- Find the correct values for the number of iterations and learning rate of the GD. You should obtain a similar plot as below and the cost about 0.87.



1.2 Linear Regression — data normalization

In this exercise we modify the previous file. We extend it by data normalization. Assume we have the data matrix (our matrix is $2 \times N$ but here we consider the more general case $(M+1) \times N$)

$$\mathbf{X}_{\text{data}} = \begin{bmatrix} 1 & \dots & 1 \\ x_1^{(1)} & \dots & x_1^{(N)} \\ \vdots & \ddots & \vdots \\ x_M^{(1)} & \dots & x_M^{(N)} \end{bmatrix} \tag{8}$$

Each of the rows corresponds to a different input variables $X_1, \ldots, X_M$. That is, we consider each entry in the vector of input variables from the training set as being a different scalar variable. E.g., the second row of $\mathbf{X}_{\text{data}}$ contains the samples from the first entry of training data vector. Each of this entries may have different mean and variance. This may lead to overflow/underflow in computations as well as makes GD to converge slower. A common approach in machine learning is to normalize the variables $X_1, \ldots, X_M$. This can be done by computing the mean μ_i and the variance σ_i^2 (or the standard deviation denoted by σ_i) of each of the variables and transforming the variables in the following way

$$X_i \leftarrow \frac{X_i - \mu_i}{\sigma_i}. \tag{9}$$

This way each of the new variables has zero mean and unit variance.

The normalization in our case can be performed by computing the mean and the variance of each row from $\mathbf{X}_{\text{data}}$ and applying the normalization transformation to each of the samples in the row. The row with ones is not normalized, as it corresponds to the bias term w_0, and is fixed by construction. After training the model with the transformed input variables, if we want to apply the model to new data, we need to perform the same transformation (with means and variances computed on the training data). That is, first we transform $\underline{x}_{\text{new}}$ just as we transformed all our training data. Next we perform prediction on the transformed version of $\underline{x}_{\text{new}}$. This is automatically done for you in the function `DataSet.plot_model(w, extend_data, norm_param)` if you provide the normalization parameters.

- Complete the function `mlcomm.nn.utils.normalize_data(x)`
  The function should normalize the rows of matrix (8) according to the above mentioned procedure. The first row is not normalized. As a second output value the function should generate a dictionary `norm_param` with two keys "mean" and "var". `norm_param['mean']` should be a column vector (`np.array of shape (M+1,1)`) containing the means used in the normalization. `norm_param['var']` should contain the variances used in the normalization. Make sure that the values for the first row (one which is not normalized) are 0 and 1 in the vector "mean" and "var" vectors, respectively.

- You should obtain exactly the same results as in the previous exercise.

1.3 Linear Regression — extending input data

In this section we will extend our model to include non-linear functions of the input data. This modification greatly improves the variety of functions which can be approximated. For such an extension the input data normalization is very helpful. Otherwise GD can be very inefficient.

- Modify the function `mlcomm.nn.utils.poly_extend_data1D(x,P)`.

Recall that the function used to output

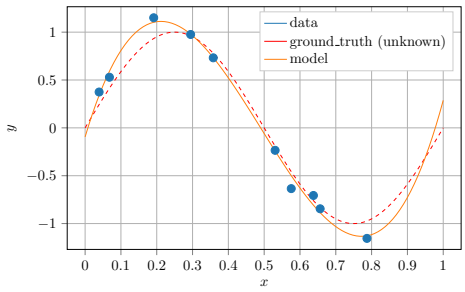
$$\mathbf{X}_{\text{data}} = \begin{bmatrix} 1 & \dots & 1 \\ x^{(1)} & \dots & x^{(N)} \end{bmatrix}. \tag{10}$$

Now, we would like to perform regression also with respect to non-linear functions of X. We will introduce polynomial terms up to degree P. Therefore the function should output now

$$\mathbf{X}_{\text{data}} = \begin{bmatrix} 1 & \dots & 1 \\ x^{(1)} & \dots & x^{(N)} \\ \vdots & & \vdots \\ (x^{(1)})^{P-1} & \dots & (x^{(N)})^{P-1} \\ (x^{(1)})^{P} & \dots & (x^{(N)})^{P} \end{bmatrix}. \tag{11}$$

Of course, $\underline{w}$ should have now $P+1$ entries, and the data normalization should be performed on the newly generated matrix $\mathbf{X}_{\text{data}}$. P should be a parameter which you can change. Note that for $P = 1$ we have the same model as before, i.e., linear.

- Modify also the wrapper `extension_wrapper(x)` so it uses the newly added polynomial extension with degree P.
- Find the parameter P, the learning rate, and the step size to get the cost function below 0.03. The result may look like this



1.4 Linear Regression — more complex models

Import the second dataset `dataset2_linreg` instead of the previous one. The objective is to minimize the cost.

- Optimize P, GD-learning rate, number of GD-iterations
- Modify the function `mlcomm.nn.utils.sin_extend_data1D(x,P)`. The goal is to try different strategy of extending the input data set. The function should produce

P harminics of the basic frequency. The output matrix should look like (comparing to the previous setup):

$$\mathbf{X}_{\text{data}} = \begin{bmatrix} 1 & \dots & 1 \\ \sin 2\pi \frac{x^{(1)}}{x_{\max}} & \dots & \sin 2\pi \frac{x^{(N)}}{x_{\max}} \\ \vdots & & \vdots \\ \sin 2\pi (P-1) \frac{x^{(1)}}{x_{\max}} & \dots & \sin 2\pi (P-1) \frac{x^{(N)}}{x_{\max}} \\ \sin 2\pi P \frac{x^{(1)}}{x_{\max}} & \dots & \sin 2\pi P \frac{x^{(N)}}{x_{\max}} \end{bmatrix}, \tag{12}$$

where $x_{\max}$ is the sample from the training data with maximum value.

- Optimize P, GD-learning rate, number of GD-iterations for this extension.