

Machine Learning for Signal Processing


[5LSL0]

Rik Vullings
Ruud van Sloun
Nishith Chennakeshava
Hans van Gorp

**Assignment: Optimum Linear and Adaptive Filters,
and Familiarisation with Pytorch**

April 2023

You may fill out your answers using the word processor of your choice, like word, or latex. Do not forget to include the relevant calculations and figures in your report.

Assignments that require you to code in python are denoted with a little  icon. You may code in any IDE you like. For your own benefit, keep the code clean, legible, and include lots of comments. Code that you create for this assignment can be repurposed for later assignments. Moreover, denote to which exercise each piece of code belongs, this makes comparing it to the answers easier.

Introduction

Adaptive filtering

The adaptive filtering techniques covered in this course attempt to find a set of weights for a Finite Impulse Response (FIR) filter $\underline{\mathbf{w}} = (w_0, w_1)^t$ that minimizes an objective function J . The objective function is chosen such that it represents the squared difference between a filtered version of an input signal $x[k]$ and a given reference signal $y[k]$, resulting in $J = E(e[k]^2)$ with $e[k] = y[k] - \underline{\mathbf{w}}^t[k]\underline{\mathbf{x}}[k]$. The final FIR filter parameters will therefore give the best possible estimate of the reference signal based on $x[k]$.

Known statistics — Wiener/GD/Newton

When the signal statistics are known, i.e., we collect the auto correlation of $x[k]$ in matrix \mathbf{R}_x and the cross correlation between $x[k]$ and $y[k]$ in the vector $\underline{\mathbf{r}}_{yx}$, we can easily solve the optimization problem. As deduced in the course, we can compute these optimal weights by $\underline{\mathbf{w}}_0 = \mathbf{R}_x^{-1}\underline{\mathbf{r}}_{yx}$. This solution is also referred to as the **Wiener filter**.

Finding the inverse of \mathbf{R}_x can be computationally expensive and we therefore require a less complex way to find $\underline{\mathbf{w}}_0$. The **Steepest Gradient Descent (GD)** algorithm approaches $\underline{\mathbf{w}}_0$ by updating in the negative gradient direction: $\underline{\mathbf{w}}[k+1] = \underline{\mathbf{w}}[k] + 2\alpha(\underline{\mathbf{r}}_{yx} - \mathbf{R}_x\underline{\mathbf{w}}[k])$, with α determining the rate of convergence. A drawback of the GD algorithm is that the filter weights do not converge uniformly. The eigenvalue spread of auto correlation matrix \mathbf{R}_x influences the rate of convergence of individual weights.

The Newton algorithm improves upon the GD algorithm by undoing the coloration of \mathbf{R}_x . Newton's method applied to our case results in the following update rule: $\underline{\mathbf{w}}[k+1] = \underline{\mathbf{w}}[k] + 2\alpha\mathbf{R}_x^{-1}(\underline{\mathbf{r}}_{yx} - \mathbf{R}_x\underline{\mathbf{w}}[k])$. Obviously, both GD and Newton's method are not always directly applicable in practice, because we may not know the signal statistics.

Unknown statistics — LS/LMS/RLS

To overcome the practical problems of GD and Newton's algorithm, we need a way to estimate the signal statistics from our input observations. Ideally, we would compute $\bar{\mathbf{R}}_x$ and $\bar{\underline{\mathbf{r}}}_{yx}$ of stationary signals as follows:

$$\bar{\mathbf{R}}_x = \lim_{L \rightarrow \infty} \frac{1}{L} \sum_{i=0}^{L-1} \underline{\mathbf{x}}[k-i]\underline{\mathbf{x}}^t[k-i] \quad (1)$$

$$\bar{\underline{\mathbf{r}}}_{yx} = \lim_{L \rightarrow \infty} \frac{1}{L} \sum_{i=0}^{L-1} \underline{\mathbf{x}}[k-i]y[k-i] \quad (2)$$

The Least Squares (LS) solution computes the Wiener filter based on the above estimates of the signal statistics. To avoid summing a large number of elements, several algorithms exist that estimate the statistics using less complex methods.

First of all, the Least Mean Squares (LMS) algorithm uses instantaneous estimates of the statistics given by: $\bar{\mathbf{R}}_x = \mathbf{x}[k]\mathbf{x}^t[k]$ and $\bar{\mathbf{r}}_{yx} = \mathbf{x}[k]y[k]$. To obtain \mathbf{w}_0 , the GD update is applied at each iteration. This results in:

$$\mathbf{w}[k+1] = \mathbf{w}[k] + 2\alpha\mathbf{x}[k]e[k] \quad (3)$$

Normalized Least Mean Squares (NLMS) extends LMS by using a normalized input. An estimate of the input signal power is maintained in $\hat{\sigma}_x^2$. Two example algorithms for updating the estimated signal power are given in the course. The NLMS update rule becomes:

$$\mathbf{w}[k+1] = \mathbf{w}[k] + \frac{2\alpha}{\hat{\sigma}_x^2} \mathbf{x}[k]e[k] \quad (4)$$

Finally, the Recursive Least Squares (RLS) algorithm is a practical version of the LS algorithm. As opposed to an infinite window, RLS uses an exponentially decreasing sample window characterized by the 'forget factor' γ . Instead of inverting $\bar{\mathbf{R}}_x$, RLS maintains a recursive estimate of its inverse $\bar{\mathbf{R}}_x^{-1}[k]$. Combined with the estimated $\bar{\mathbf{r}}_{yx}[k]$, the final filter weights are given by $\mathbf{w}[k] = \bar{\mathbf{R}}_x^{-1}[k]\bar{\mathbf{r}}_{yx}[k]$. The recursive update rule for the estimates of the signal statistics is given by:

$$\mathbf{g}[k+1] = \frac{\bar{\mathbf{R}}_x^{-1}[k]\mathbf{x}[k+1]}{\gamma^2 + \mathbf{x}^t[k+1]\bar{\mathbf{R}}_x^{-1}[k]\mathbf{x}[k+1]} \quad (5)$$

$$\bar{\mathbf{R}}_x^{-1}[k+1] = \gamma^{-2}(\bar{\mathbf{R}}_x^{-1}[k] - \mathbf{g}[k+1]\mathbf{x}^t[k+1]\bar{\mathbf{R}}_x^{-1}[k]) \quad (6)$$

$$\bar{\mathbf{r}}_{yx}[k+1] = \gamma^2\bar{\mathbf{r}}_{yx}[k] + \mathbf{x}[k+1]y[k+1] \quad (7)$$

Known statistics

Consider the general filter optimization scheme as depicted in Fig. 1.

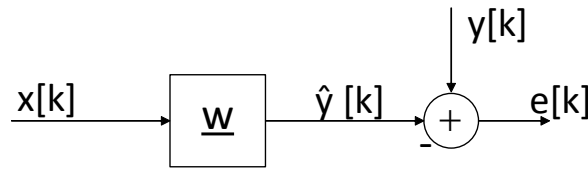


Figure 1: MMSE model

In this part of the assignment you have to calculate and design a three-tap FIR filter with $\mathbf{w} = (w_0, w_1, w_2)^t$, by minimizing the MSE $J = E\{e^2[k]\}$. The following statistics are given:

$$\mathbf{R}_x = \begin{pmatrix} 5 & -1 & -2 \\ -1 & 5 & -1 \\ -2 & -1 & 5 \end{pmatrix} ; \mathbf{r}_{yx} = \begin{pmatrix} 1 \\ 5.3 \\ -3.9 \end{pmatrix}.$$

Exercise 1: Wiener Filter

- (a) **[1 pt]** Calculate by hand an expression for the MSE $J = E\{e^2[k]\}$ as a function of the three weights w_0 , w_1 , and w_2 . Next, find the filter weights w_0 , w_1 , and w_2 that minimize the MSE J by setting $\partial J/\partial w_0 = 0$, $\partial J/\partial w_1 = 0$, and $\partial J/\partial w_2 = 0$. Show your calculations and give the numerical values of $\underline{\mathbf{w}}$.
- (b) **[1 pt]** Find an expression for the correlation $\underline{\mathbf{r}}_{xe}$ between $\underline{\mathbf{x}}[k]$ and $e[k]$. What should $\underline{\mathbf{r}}_{xe}$ be when $\underline{\mathbf{w}}$ equals the optimal Wiener filter $\underline{\mathbf{w}}_o$? What does this mean?
- (c) **[1 pt]** In practical cases, \mathbf{R}_x and $\underline{\mathbf{r}}_{yx}$ may be unknown. How can you estimate these statistics from a set of observations $\underline{\mathbf{x}}[k]$ and $y[k]$? Which tradeoff do you have to make?

Exercise 2: Steepest Gradient Descent

Computing the optimal Wiener filter involves inverting \mathbf{R}_x . To avoid the matrix inversion, one can use an iterative approach. The Steepest Gradient Descent (GD) algorithm is such an algorithm.

- (a) **[1 pt]** When does the GD algorithm reach steady-state? In other words, when the GD reaches steady-state, what is the value of $\underline{\mathbf{w}}[k]$?
- (b) **[1.5 pt]** Give the range of α which makes the GD algorithm stable. Calculate the eigenvalues of \mathbf{R}_x by hand (i.e. show the derivation). Hint: one of the eigenvalues $\lambda = 7$.
- (c) **[1 pt]** ♣ Implement the filter update rule of the GD algorithm in Python. Take the following steps to do so.
 - Download the file `assignment1_data.csv` from Canvas. This file contains generated samples of both $\underline{\mathbf{x}}[k]$ (column 1) and $y[k]$ (column 2).
 - Initialize the filter
 - Implement the adaptive filter and perform N iterations of this adaptive filter, where N is the number of samples in the dataset. During each iteration, update the filter coefficients.
 - Plot the trajectory of the filter coefficients as they evolve, together with a contour plot of the objective function J . Plot the convergence for two of the three filter coefficients $\underline{\mathbf{w}}$: w_0 and w_1 to avoid having to create a 3D plot. For generating the contours, choose $w_2 = -0.5$. **NOTE: for other exercises in this assignment, use the same value of w_2 and every time plot the convergence of w_0 and w_1 when being asked.**
 - Use a value of α that ensures a smooth and stable convergence of $\underline{\mathbf{w}}$. Comment on the convergence of w_0 and w_1 .

Exercise 3: Newton's Method

Input coloration influences the performance of the GD algorithm. To solve this issue, the Newton algorithm performs pre-whitening of the input signals.

- (a) **[1 pt]** Show that Newton's method makes all filter weights converge at the same rate.
- (b) **[1 pt]** For which α is Newton's method stable?

- (c) **[0.5 pt]** ♣ Implement the Newton filter update rule in Python. Again use an α that gives smooth convergence. Use the code from the previous exercise as starting point and modify it such that the filter uses the Newton update rule. Run your code and comment on the convergence of the filter parameters.

Unknown statistics

Both the GD and Newton method assume that \mathbf{R}_x and \mathbf{r}_{yx} are known. However, in practical cases these are often unknown. The Least Mean Squares (LMS) and the Normalized LMS (NLMS) algorithms use an instantaneous estimate of the signal statistics.

Exercise 4: LMS and NLMS

- (a) **[1 pt]** ♣ Implement the LMS filter update rule, where you use your code from Assignment 1a as starting point. Again, use a filter length of 3 and an appropriate α . What trade-off do you make when choosing α ? Include the resulting plot and the most relevant line(s) of code in your answer form
- (b) **[0.5 pt]** ♣ Repeat for Normalized LMS.

Exercise 5: RLS

The RLS algorithm attempts to whiten the input using estimates of the signal statistics.

- (a) **[1 pt]** Explain in your own words how RLS is related to the previous adaptive algorithms (LS/GD/Newton/LMS/NLMS). How does RLS solve the practical issues of the corresponding algorithm?
- (b) **[1 pt]** ♣ Implement the RLS update rule in Python. Use $\gamma = 1 - 10^{-4}$ and include the plot of convergence of the filter coefficients in your answer form. How does γ influence the convergence behavior?
- (c) **[1 pt]** Compare the advantages and disadvantages of LMS, NLMS, and RLS in terms of computational complexity and accuracy. Try to rank them (#1 is the best, #3 the worst). Give a brief explanation.

Familiarisation with Pytorch

In this assignment, you will implement a basic neural network. This is for the purpose of performing denoising on given data (MNIST). You should be able to complete this assignment using what is given here, the tutorials, and pytorch's web-pages.

Neural Networks

There are many forms of Neural Networks (NN), which are used for varying applications. In this assignment, a Feed Forward Neural Network (FFNN) is used. These networks are comprised of "neurons", which can be described as,

$$y = f \left(\sum_{i=0}^n w_i x_i + b \right)$$

where w_i is a weight multiplied to the input x_i , and b is a bias added to the product. Finally, an activation function f is applied, giving the output, y .

Neural networks of this fashion can be used as universal function approximators. As such, you will now be asked to do just that. Familiarise yourself with Pytorch and Python, all while testing the abilities of neural networks to operate as function approximators.

It might be good to explain at this point that you do not necessarily have all of the information or knowledge to fully understand the technical aspects of what you are expected to do in this assignment. But the goal is for you to acquire this knowledge in the process of completing this assignment and the rest of the course. All components of this assignment will also be covered in future classes of this course, in more detail.

As a first step you will need to install Pytorch so that you can exploit this powerful library for designing, implementing, and training your neural network. To install Pytorch, go to <https://pytorch.org/get-started/locally/> and follow the instructions.

You should be aware that using both Pytorch and TensorFlow at the same time can throw errors and is not recommended.

The aim is to use a FFNN to denoise MNIST data, a dataset of hand written numbers. You have also been provided with a pytorch data loader for the data that we will use in this assignment. It will provide pairs of clean digits, and the corresponding noisy digit.

Note: None of the questions below require an explanation of more than 2/3 paragraphs, in addition to any graphs that may be required.

Exercise 6

- (a) **[0.5 pt]** 🛠 Import the data using the dataloader. After importing it, plot a sample of the dataset, to visualise what the data actually looks like. The image should look something like the following:

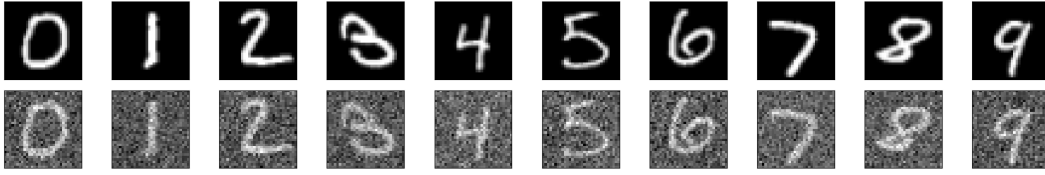


Figure 2: An illustration of the data.

Here, the top row represents the clean MNIST images, while the bottom row is the corresponding noisy images.

- (b) **[1 pt]** ♣ We can now start building the neural network!

Build a fully connected network with linear layers, to approximate a function that can denoise the input data. Here, you are simply building something that can map the inputs \mathbf{y} to the targets \mathbf{x} . Set no activation functions. Moreover, make sure to first vectorize the 2D image so that it can be used as input for a set of linear layers. Afterwards, convert it back to a 2D image.

(Hint: A multi layered network with multiple neurons per layer might work better. While a single neuron network is too small, a network with more than 10 layers might be too much.)

A network that is highly complex can introduce its own set of problems. What problems, other than computational and memory load, can occur due to a too complex network architecture?

- (c) **[0.5 pt]** ♣ For the network to start training, we also need an optimiser.

What is the goal of an optimiser within a Neural Network?

For the implementation, start with using the SGD (Stochastic Gradient Descent). This has similarities to the Steepest Gradient Descent algorithm. Start with a small learning rate to begin with.

What are the similarities and differences between the two optimisers?

- (d) **[0.5 pt]** ♣ To ensure that your model can actually run, make a prediction with the untrained network using the **test set** (after splitting your data into the training-validation-test sets).

Plot the prediction alongside the test data. Comment on the prediction of the untrained model. Why does it look the way it does?

- (e) **[1 pt]** ♣ Now, all that's left to do is to actually train your model! During training, make sure that you save the weights from your model to a folder, you may look up the code on pytorch's website. At this point, you're also ready to plot the loss from the training process. Both training losses and the testing losses.

Plot the training loss and the testing loss in one graph, with the proper labels (and a legend).

How could you utilise this information to assess the quality of your model?

- (f) **[1 pt]** ♣ Make a prediction on the test set using your trained model. Compare this to the prediction made on the untrained model.

Comment on your results.

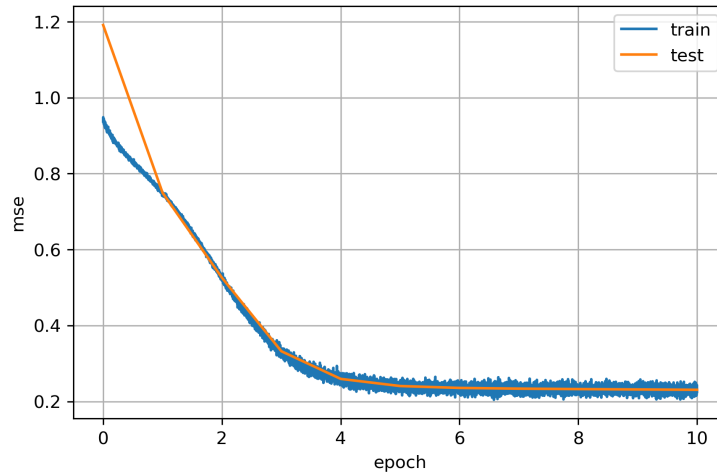


Figure 3: An example of what the training and validation losses look like after 10 epochs. Not to say that you should expect a similar graph from your training process.

Exercise 7

Now, we would like you to build upon what you did in the previous exercise. One of the reasons your network might not have learnt to approximate the desired function might be due to the architecture of the network, as it requires some degree of non-linearity. We are going to introduce this non-linearity to the network in the form of an activation function.

- (a) **[1 pt]** ♣ Build an activation function (ReLU), and include it with the model. The definition for a ReLU is given as,

$$f(x) = \max(0, x) \quad (8)$$

You may use the same network as before, but now use your own activation function as the activation function (Do not use the ReLU activation function given by Pytorch)!

Plot the output from this activation function. Does it match what you expect? Where in the network does the activation function have to be implemented? Include your custom built activation in the network. Also include the code that you have written.

- (b) **[1 pt]** ♣ A second major factor can be the optimiser that you chose to perform this regression. While SGD does as advertised, there are more sophisticated optimisers out there.

For the next round of training, pick a more sophisticated optimiser, and justify your choice. What are the problem that can occur with SGD and how are these resolved by more sophisticated optimizers?

- (c) **[0.5 pt]** ♣ As with exercise 1d, make a prediction using the untrained network using the test data.

Comment on the prediction of the untrained network.

- (d) **[1 pt]** ♣ Train this network, and ensure to save the weights from the training, so that you can load it in later to make prediction.

Plot train and validation losses, like you did in exercise 1e.

At this point, it is good to reevaluate if the network architecture you have chosen is appropriate for the problem you are trying to solve.

- (e) **[1.5 pt]** ♣ Make a prediction on the test set using your trained model. Compare the prediction made by the first trained model (from Exercise one), to the prediction made by the second trained model (from Exercise two). Also comment on the differences between the two models that you have implemented.
- (f) **[1 pt]** Based on this comparison, do you think the linear model would benefit from additional layers? Justify your answer.