Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Signal and Information Processing Laboratory
Institut für Signal- und Informationsverarbeitung

Fachpraktikum Signalverarbeitung

SV6: Polynomial Regression and Neural Networks

1 Introduction

Consider the situation in Figure 1 where some unknown function $f(\cdot)$ is embedded in a black box together with a noise source Z. For some given value x we don't have access to the actual function value $v = f(x)$ but observe a noisy value $y = f(x) + Z$.

Assume now that we want to construct a function $\varphi(\cdot)$ which should closely match the unknown function $f(\cdot)$. In this approach we apply m values $x^{(1)}, x^{(2)}, \ldots, x^{(m)}$ and observe the corresponding outputs $y^{(1)}, y^{(2)}, \ldots, y^{(m)}$. From this *sample data* we try to adjust the parameters in $\varphi(\cdot)$.

When we somehow have built our function $\varphi(\cdot)$ we want to use it to predict $v = f(x)$ by an estimated value $\hat{v} = \varphi(x)$ for some x.

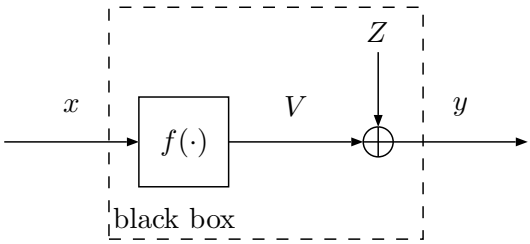

Figure 1: Model for Measuring a Function $f(\cdot)$.

In this experiment we will describe $\varphi(\cdot)$ as a polynomial or as a neural network and try to find the respective coefficients such that the match is close and $\varphi(\cdot)$ has good generalization properties.

2 Polynomial Regression

At first we try to approximate $f(\cdot)$ in Figure 1 by a polynomial $\varphi(\cdot)$ of degree n:

$$\varphi(x) = a_0 + a_1 x + a_2 x^2 + \ldots + a_n x^n$$

It is the aim now to find the coefficients $a_0, a_1, \ldots, a_n$. In principle this can be achieved by minimizing the *average squared error*

$$ASE = \frac{1}{m} \sum_{k=1}^{m} (y^{(k)} - \varphi(x^{(k)}))^2 \tag{1}$$

which leads to a so-called *least squares problem*.

The coefficients can be found by solving the following equation:

$$\mathbf{U}^T\mathbf{U}\,\boldsymbol{a} = \mathbf{U}^T\boldsymbol{y} \quad \text{where ,} \tag{2}$$

$$\mathbf{U} = \begin{pmatrix} 1 & x^{(1)} & \left(x^{(1)}\right)^2 & \cdots & \left(x^{(1)}\right)^n \\ \vdots & \vdots & \vdots & & \vdots \\ 1 & x^{(m)} & \left(x^{(m)}\right)^2 & \cdots & \left(x^{(m)}\right)^n \end{pmatrix}, \quad \boldsymbol{a} = \begin{pmatrix} a_0 \\ \vdots \\ a_n \end{pmatrix} \quad \text{and} \quad \boldsymbol{y} = \begin{pmatrix} y_0 \\ \vdots \\ y_m \end{pmatrix} .$$

It is expected that as we increase the order n the better the fit will be, i.e. the ASE will decrease as we increase n. In particular for $n+1 = m$ we can achieve an ASE of 0. This is because we can fit a polynomial of order n through $n+1$ points.[1]

However, increasing n too much can result in *over-fitting*. We will explore this effect later.

3 Neural Networks

A neural network is a special family of functions.[2] In this experiment we consider a so-called *two-layer perceptron*. To be a little more general, we assume that in Figure 1 the input is a vector with two components $x = (x_1, x_2)$. In this case we define $\varphi(\cdot)$ as:

$$\varphi(x) = g_2\left(b_0 + \sum_{l=1}^{L} b_l \cdot \underbrace{g_1\left(a_{l0} + a_{l1}x_1 + a_{l2}x_2\right)}_{r_l(x)}\right) \tag{3}$$

The parameter L is the number of internal dimensions of the perceptron, $a_{l0}, \ldots a_{l2}$ are coefficients in the first layer and $b_0, \ldots b_l$ are coefficients in the second (the output) layer. $g_1(\cdot)$ is a nonlinear function which can be chosen.

Figure 2 shows the block diagram of the function in Equation (3) for the case $L = 3$. The network thus first forms three so-called *hidden variables* $r_l(x)$ by linearly combining the input (x_1, x_2) in L different ways and passing the results through the nonlinear function $g_1(\cdot)$. The output $\hat{v}$ is then formed by a linear combination of the hidden variables r_l and taking the output function $g_2(\cdot)$.

A popular choice for $g_1(\cdot)$ is the hyperbolic tangent $\tanh(t) = \frac{e^t - e^{-t}}{e^t + e^{-t}}$ .The output function $g_2(\cdot)$ is often a pure linear function $g_2(t) = t$ in which it can as well be omitted. It has been shown that for this choice the neural network can represent any continuous function $f(\cdot)$ to arbitrary accuracy provided L is large enough.

For some choice of L, $g_1(\cdot)$ and $g_2(\cdot)$ the question is now how to find the coefficients $a_{l0}, \ldots a_{l3}$ and $b_0, \ldots b_l$. If $g_2(\cdot)$ is linear then – for fixed $a_{l0}, \ldots a_{l3}$ – finding $b_0, \ldots b_l$ is again a least squares problem, but the coefficients of the first layer have to be found by nonlinear methods.

These usually imply *training* the coefficients which is quite different from solving a least squares problem. Assume some initial values for the coefficients. We then can ask the question: How does the ASE change when we change the coefficients? This leads to a concept called *backpropagation*, which computes the derivatives of the error with respect to the coefficients[3] and adjusts them such that the error decreases. This is done iteratively until some criteria is achieved. The result need not be optimal however.

Despite the ability of the network to represent any function almost perfectly, over-fitting can occur easily with backpropagation for large L.

[1] E.g. a line (polynomial of order $n = 1$) can be fitted through 2 points.

[2] which was originally inspired by the nervous system in animals, hence the name.

[3] For this to be possible we have to know the derivative of the nonlinearities $g_1(\cdot)$ and $g_2(\cdot)$.

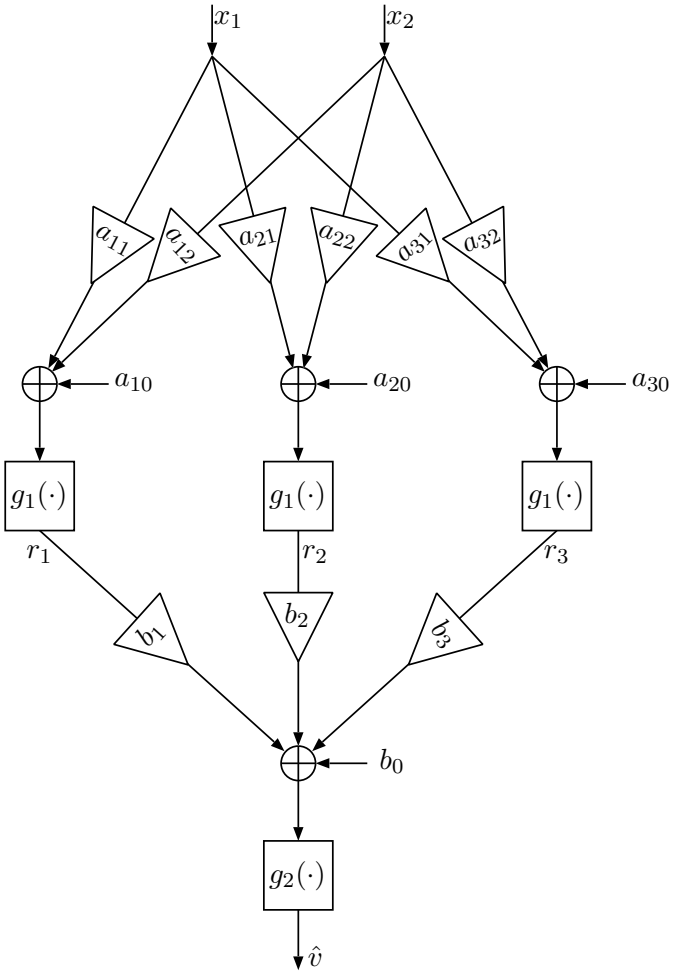

Figure 2: Two-Layer Perceptron.

4 Over-Fitting

The methods described above are both sensitive to what is called *over-fitting*. By training the function $\varphi(\cdot)$ with m data points, it might reconstruct the values at these points very accurately, but fail completely on new data. The function $\varphi(\cdot)$ has just "memorized" the sample data but has poor *generalization* behavior.

It seems that the following factors have an impact:

- The relation between m of $f(\cdot)$ and the number of adjustable parameters in $\varphi(\cdot)$ – the *order* of the model $\varphi(\cdot)$. Fitting a model with many parameters to few sample points generated by a simple function can result in over-fitting.
- The choice of the abscissae $x^{(k)}$ of the sample data points.
- The method of finding the parameters in $\varphi(\cdot)$. If this method considers that $\varphi(\cdot)$ should have a high generalization capabilities, over-fitting can be prevented.
- The target function $f(\cdot)$.

In the case of polynomial fitting, choosing n lower than $m-1$ is crucial. Furthermore choosing $x^{(k)}$ such that they are more crowded toward the endpoints of the interval of interest reduces over-fitting. Given a target interval $[\alpha, \beta]$ on the x-axis, the optimal

choice for $x^{(k)}$ has been shown to be:

$$x^{(k)} = \frac{1}{2}(\alpha+\beta) + \frac{1}{2}(\alpha-\beta)\cos\left(\frac{\pi(2k-1)}{2m}\right), \quad k = 1, 2, \ldots, m. \tag{4}$$

For neural networks one approach is called *regularization*. The cost to be minimized is not simply the ASE but also includes the sum of the squared coefficients.

$$\text{Cost} = \lambda\ ASE + (1-\lambda)\underbrace{\left(\frac{1}{L}\sum_{l=0}^{L} b_l^2 + \frac{1}{L}\sum_{l=1}^{L}\frac{1}{K}\sum_{k=0}^{K} a_{lk}^2\right)}_{\text{“average squared coefficient”}}, \quad 0 < \lambda < 1 \tag{5}$$

In this way the absolute value of the coefficients are kept low on average.

A general approach to finding the order of $\varphi(\cdot)$ is as follows. Use a portion of the samples for training and the remaining ones serve as test for the generalization capabilities of the trained $\varphi(\cdot)$. Then different model orders can be tried and generalization can be assessed.

References

H.-A. Loeliger, *Introduction to Estimation and Machine Learning*, ETH, Lecture Notes, 2020.

J.G. Proakis und D.G. Manolakis, *Digital Signal Processing, 4th Edition*, Prentice Hall, 2006.

5 Experiments

Copy `/home/isistaff/glf/fachprak_isi/SV6` to your home directory:
(`cp -irL /home/isistaff/glf/fachprak_isi/SV6 ./`)
In these experiments you will use the Matlab command line and complete some Matlab script files (.m files) which are located in the `SV6/matlab` directory. Furthermore, we will have a look at the Matlab “Deep Learning Toolbox”. Start Matlab from a shell by switching into the directory `SV6/matlab` and invoking `matlab &`.

1. The script `funny.m` provides a funny target function $f(\cdot)$ for our black box in Figure 1. Have a look at it by typing on the Matlab command line:
   `> x = linspace(-1.5, 1.5); plot(x, funny(x));`
   `funny` consists of pieces of an inverse quadratic, linear and sinusoidal function.

2. The noise Z in Figure 1 is also implemented. This is done by giving the *noise variance* as a second parameter:
   `> plot(x, funny(x, 0.005), '.');`

5.1 Polynomial Curve Fitting

The script `mypolyfit.m` generates sample data of the `funny` function for a given noise variance `var`. The column vector `xS` holds the linearly spaced abscissae $x^{(k)}$ and the column vector `yS` holds the corresponding values $y^{(k)} = f(x^{(k)}) + Z_k$. The script plots the sample data in blue, the actual funny function $f(\cdot)$ in black and the polynomial function $\varphi(\cdot)$ in red.

3. Implement the solution to the system of equations given in Equation (2) in `mypolyfit.m`. You can test your implementation by typing `> mypolyfit();`

   Hints: The construction of the matrix $\mathbf{U}$ is already implemented in the script.
   The standard routine in Matlab for solving a system of equations is the backslash operator '\'. For information on its usage type `> help mldivide`.
   The transposition $\mathbf{A}^T$ is written in Matlab as `A'`.
   An even simpler way to solve the least squares problem is to use Matlabs backslash operator for the over-determined equation system $\mathbf{U}\boldsymbol{a} = \boldsymbol{y}$.[4]

4. Run the script for different polynomial orders `n`, number of samples `m` and noise variances `var`. All three parameters can be passed to the function, e.g:
   `> mypolyfit(20, 100, 0.001);`

The following observations can be made:

- The greater m and the lower the variance, the better the fit. More information about the funny function is obtained.
- If n is too low the fit will be poor, e.g. if $n = 1$ then $\varphi(\cdot)$ is a straight line fitted to the sample data.
- If n is too high then over-fitting occurs, especially if m is low, e.g $n = 27$, $m = 50$. If $n + 1 \geq m$ and m is moderate, the fit will go through the sample points but otherwise be very bad, e.g. $n = 12$, $m = 9$. The former effect disappears for larger m. This is because the rank[5] of the matrix $\mathbf{U}$ does not increase arbitrarily but – in this specific example – saturates at about 34. For even larger values m numerical problems can arise depending on the solution of Equation (2).

The script `autopolyfit.m` does the polynomial fitting successively with increasing order $n = 1, \ldots, 50$.[6]

5. Try the following: `> autopolyfit(25)` to see how polynomials with increasing orders are fitted to $m = 25$ data points.[7]

The upper plot in the figure shows the fit in a similar way as `mypolyfit` did. The lower plot shows ASE_1 in blue and ASE_2 in black for increasing order n. While ASE_1 is taken with respect to the sample data as in Equation (1), ASE_2 is the ASE with respect to a huge number of noiseless samples and visualizes how the fit will perform on new data. At the end `autopolyfit` makes a choice on n such that $n \leq m - 1$ and ASE_2 is minimal. The result is shown as a green line in the upper plot and the choice is written in the title of the lower plot.
Notice that the occurrence of over-fitting can be detected in the rise of ASE_2. We try now to lessen the over-fitting even for few samples m by choosing the sample data points as in Equation (4).

6. Locate the generation of the samples abscissae `xS` in `autopolyfit` and replace it by an implementation of Equation (4).

[4] Note that different implementations of the solution to this equation can yield quite differing results due to differing numerical stability.

[5] The number of independent columns or rows, whichever is smaller.

[6] `autopolyfit.m` invokes Matlabs `polyfit.m` function, which essentially uses the same procedure as we did, but makes use of the QR-decomposition to achieve higher numerical stability.

[7] If you want to have a slower or faster display then edit the `pause(0.2);` statement (last line in the `for`-loop) in `autopolyfit.m` to your needs.

Hint You can directly implement a vectorized version of the equation by noting that the Matlab statement `(1:m)` produces a vector $(1, 2, \ldots m)$.

7. Again try: `> autopolyfit(25)`. Note the new spacing of the sample data points. Try other choices of m.

For high values m the spacing of the sample data $x^{(k)}$ does not seem to play a great role anymore.
Next we are going to see whether we still can do the fit with noisy observations. `autopolyfit` takes the noise variance as a second argument and returns ASE_2 for the chosen polynomial order n.

8. Vary `m` and `var` but keep the ratio `m/var` constant, e.g.
   `> autopolyfit(50, 0.005)`, `...(500, 0.05)`, `...(5000, 0.5)`.

Note that the resulting minimal ASE_2 and the visual fit are similar in all cases similar, even for large noise variances. Also notice that ASE_1 approaches the noise variance as n increases, whereas ASE_2 usually takes on much smaller values.
You might have observed that for orders $n \gtrsim 38$ both ASE_1 and ASE_2 sometimes suddenly increase or fluctuate strongly. This is because the solution to the equation system (2) is not numerically stable.[8]

5.2 Regression with a Two-Layer Perceptron

The script `myneuralfit.m` implements a two-layer perceptron as in Figure 2, but with only one input and a flexible number L of hidden variables. It takes L, the number of sample data points m and the noise variance as arguments. Training is done using a so-called *conjugate gradient* algorithm which relies on the error gradients obtained from backpropagation.

9. Look at how the neural network is implemented and trained in `myneuralfit.m`. Note the following three statements which make use of Matlab's neural network toolbox: `newff(...)` creates a *feed forward* network – a two-layer perceptron in our case, `train(...)` trains the network coefficients and `sim(...)` calculates the network response to some input.

As the network is trained, the ASE (which is to be minimized) is displayed. Once the training is done[9], the network response is displayed in red together with the sample data points (blue) and the actual values of the funny function (black). If a fourth argument of -1 is given then a polynomial fit with order $n = L$ is displayed in green.

10. Run `> myneuralfit(20, 100, 0.001, -1);` (compare with step number 4). This might take some seconds.[10]

Both the polynomial and the neural network fit visually perform quite similar. Since m is large compared to L (and n) the fit is ok.
Now we make the fitting task more difficult by choosing the abscissae $x^{(k)}$ of the sample points randomly, i.e. uniformly distributed over the interval of interest.

[8] Normally Matlab issues warnings under such conditions but these have been suppressed in this example.

[9] The training can be stopped early by clicking on the "Stop Training" button in the figure.

[10] As with all neural network training methods, the result depends on the initial values of the weights. Since these are usually chosen somewhat randomly, individual runs might lead to bad results. Remember: there is no guarantee for optimality. In rare occasions you might have to run scripts twice to get a good impression.

11. Change the generation of `xS` in the script by commenting out the respective line with `%` and uncommenting the next line.

12. Fitting now to few sample data points can result in over-fitting.
    Run e.g: `> myneuralfit(20, 30, 0.001, -1);`. Experiment to get a feeling of how the network and the polynomial fit are behaving.

13. Reduce the effect of over-fitting by decreasing the model order:
    `> myneuralfit(9, 30, 0.001, -1);`.

14. Actually, fitting to many noisy sample points results in over-fitting too.
    `> myneuralfit(40, 800, 0.1, -1);`.

Next we make use of regularization as in Equation (5), more precisely we use *Bayesian regularization* which chooses the parameter λ automatically. We do this by applying a different training method from the Matlab neural network toolbox.

15. On the line where the network is created change the last parameter given to the `newff` function to `'trainbr'`. This tells Matlab to use the Bayesian regularization based training function `trainbr` instead of the conjugate gradient based training function `traincgf`.

16. Now try again: `> myneuralfit(20, 30, 0.001, -1);` and
    `> myneuralfit(40, 800, 0.1, -1);`.

Notice that training now takes longer but over-fitting is reduced considerably.
As a last experiment we do a two dimensional fitting and let $f(\cdot)$ be the Matlab function `peaks()`. The network is the one shown in Figure 2 with two inputs (x and y coordinate), one output (z coordinate of a point) and $L = 50$ hidden variables. This time the sample data does not have any additive noise.

17. Have a look at the peaks (close open figures first):
    ```
    > x = linspace(-1, 1, 50); y = x;
    > [xx, yy] = meshgrid(x, y); surf(xx, yy, peaks(50));
    ```

18. Train the network and display the result:
    `> neuralpeaks();`

19. In the `neuralpeaks.m` script change the training function back to `'traincgf'`, run again and look what happens.

Congratulation, you have reached the end of the experiments. If there is still time left, feel free to play around with the functions provided in the Matlab "Deep Learning Toolbox".