4. APPLICATIONS

In this chapter we will look at some applications of the block pseudoinversion formula (2.22). The advantage of the formula is that when processing any data, you can independently (in parallel) collect various data (matrix blocks) and calculate pseudo-inverses to them independently of each other, therefore, this formula is well suited in the case when data arrives sequentially with some latency, or when processing data in parallel. Recall that formula (2.22) only works in the case of linearly independent columns, however, in practice this is the most common case.

4.1. Restoring noisy signals

The data processing problem often reduces to solving an overdetermined system of linear algebraic equations (see [48], [49], [50], [51]). In the general case, for overdetermined systems, the minimum norm vector is taken as a solution, i.e. pseudo-solution. Often a pseudo-solution is found using the matrix pseudo-inversion method, which uses singular value decomposition, but this method is not always optimal. In most cases, it is assumed that the received data is received in full, but in practice situations arise when the data arrives with some delay or is processed sequentially. In such a situation, columns are gradually added to the initial matrix of the system, and to find a pseudo-solution, a recalculation of the resulting matrix is required at each stage. In this case, finding a pseudoinverse matrix using the singular value decomposition method will not be optimal, since the solution in this way has a third order of complexity $O(n^3)$. Here, when the next block of data arrives, we already know the pseudo-inverse matrix to the block at the previous stage, so it is proposed to use the block pseudo-inversion formula. Despite the fact that the formula also uses pseudo-inversion, it is not the full matrix that needs to be pseudo-inverted, but only the resulting blocks, which reduces the number of calculations. In addition, the formula is applicable for receiving signals from different sources, that is, it allows you to find a pseudo-inverse matrix when adding several blocks. In a particular case, the proposed

algorithm is one step of the Greville algorithm (or one step of the Kaczmarz algorithm ([51], [52]) and can be used to train neural networks (see [50]) or to solve the problem of identifying neighborhood systems with a history ([19]).

Let Y be the incoming signal and X the original signal. X, Y can be either vector or matrix quantities. Let also, the incoming signal is obtained by the influence of any interference on the original signal according to the following formula:

$$Y = HX,$$

where H – matrix (for example, in the case of a one-dimensional electrical signal, H will be sets of vectors composed of source impulse response values over a certain period of time). Let us further assume that the incoming signal arrives to us in portions, for example, at the first stage we received part of the incoming vector Y_1 and parts of the transformation matrix of the original vector H_1:

$$H_1 X_1 = Y_1.$$

At the second stage, taking the second part of the signal we obtain the equation

$$\left[H_1 H_2\right]\begin{bmatrix} X_1 \\ X2 \end{bmatrix} = \begin{bmatrix} Y_1 \\ Y_2 \end{bmatrix}.$$

And so on, for n stages:

$$\left[H_1 H_2 ... H_n\right]\begin{bmatrix} X_1 \\ ... \\ X_n \end{bmatrix} = \begin{bmatrix} Y_1 \\ ... \\ Y_n \end{bmatrix}.$$

Assume, that $[H_1 H_2 ... H_n]$ has a full column rank. The task is to gradually restore part of the signal $\left[X_1^T X_2^T ... X_n^T\right]^T$ i-th stage. When solving this problem, we will use the formula for block pseudoinversion of full-rank matrices obtained in (2.22). Now let's describe the complete algorithm.

At the first stage we have

$$\left[H_1 H_2\right]\begin{bmatrix} X_1 \\ X_2 \end{bmatrix} = \begin{bmatrix} Y_1 \\ Y_2 \end{bmatrix} \text{ и } \begin{bmatrix} X_1 \\ X_2 \end{bmatrix} = \left[H_1 H_2\right]^+ \begin{bmatrix} Y_1 \\ Y_2 \end{bmatrix}.$$

To find $[H_1 H_2]^+$ use the formula (2.22):

$$\left[H_1 H_2\right]^+ = \begin{bmatrix} E & H_1^+ H_2 \\ H_2^+ H_1 & E \end{bmatrix}^{-1} \begin{bmatrix} H_1^+ \\ H_2^+ \end{bmatrix}, \tag{4.1}$$

then the reconstructed signal is found by the formula

$$\begin{bmatrix} X_1 \\ X_2 \end{bmatrix} = \begin{bmatrix} E & H_1^+ H_2 \\ H_2^+ H_1 & E \end{bmatrix}^{-1} \begin{bmatrix} H_1^+ \\ H_2^+ \end{bmatrix} \begin{bmatrix} Y_1 \\ Y_2 \end{bmatrix}. \tag{4.2}$$

When further parts of the signal arrive, formula (4.2) is used recursively (the matrix $[H_1 H_2]$ will be the first block for the matrix $[H_1 H_2 \vdots H_3]$). If you need to add several blocks at once, if, for example, the signal comes from different sources, then formula (4.2) will take the form

$$\left[H_1 ... H_N\right]^+ = \left[\begin{bmatrix} H_1^+ \\ ... \\ H_N^+ \end{bmatrix} \left[H_1 ... H_N\right]\right]^{-1} \begin{bmatrix} H_1^+ \\ ... \\ H_N^+ \end{bmatrix}$$

where we get the original vector

$$\begin{bmatrix} X_1 \\ ... \\ X_n \end{bmatrix} = \left[\begin{bmatrix} H_1^+ \\ ... \\ H_N^+ \end{bmatrix} \left[H_1 ... H_N\right]\right]^{-1} \begin{bmatrix} H_1^+ \\ ... \\ H_N^+ \end{bmatrix} \begin{bmatrix} Y_1 \\ ... \\ Y_n \end{bmatrix}$$

Finding a pseudoinverse matrix using formula (4.2) has the same order of complexity as finding it using singular decomposition or the Hermitian formula (see [53], [54]). However, we know the pseudo-inverse matrix for the block H_1 and by dividing into blocks (reducing the dimension of matrices) we obtain a complexity estimate of $O(\frac{7}{2}n^3)$..

The proposed algorithm works if the matrix has full column rank and the number of rows is greater than or equal to the number of columns (in the opposite case, the formula is easily transposed), however, in signal reconstruction problems this requirement is almost always met [55]. In addition, computational experiments show that on average formula (4.2) is 20% more stable compared to the singular value decomposition method or the Hermitian formula, and the algorithm copes well even

with poorly conditioned matrices. The algorithm is also suitable for the case when the noise is not static. Then we have the opportunity to recalculate the pseudoinverse matrix when any noisy part changes.

4.2. One-dimensional signal reconstruction

Let us give an example of applying the algorithm in a one-dimensional case. In the one-dimensional case, the desired signal *x(t)* is determined by the integral convolution equation (see [48], [49])

$$y(t) = \int_0^t h(t-\tau)x(\tau)d\tau, \tag{4.3}$$

where $h(t)$ – filter impulse response, $y(t)$ – output signal. Moving on to the integral sum

$$y(t) = \sum_0^{K-1} Th(t-kT)x(kT), \tag{4.4}$$

where T – an interval step, K – number of intervals, k – interval number. Let's denote $H(t) = Th(t-kT)uX = x(kT)$, get

$$y(t) = sum_0^{K-1} H(T)X.$$

Denoting $H = H(H_1 H_2 ... H_K)$ we obtain a system of linear equations

$$Y = HX. \tag{4.5}$$

Then the original signal is restored using the formula

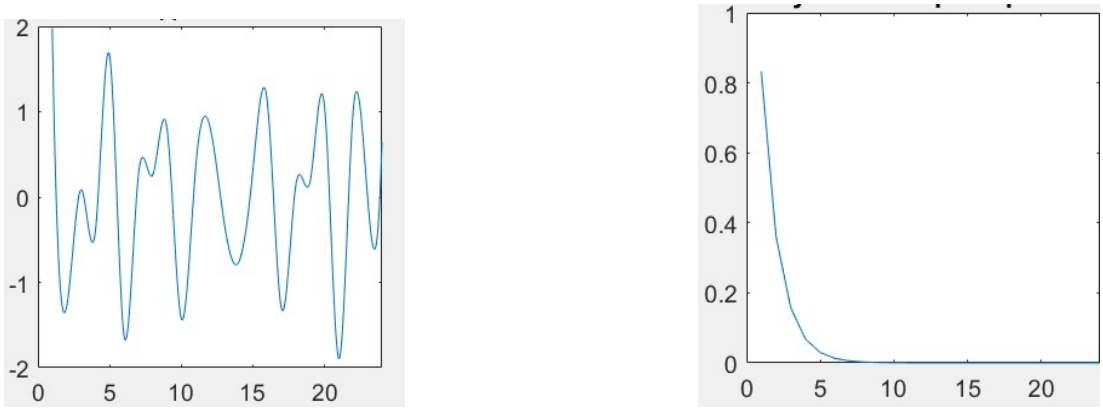
$$Y = H^+ X. \tag{4.6}$$

Matrix H has the form:

$$H = T\begin{bmatrix} h_0 & 0 & 0 & 0 \\ h_1 & h_0 & 0 & 0 \\ \dots & \dots & \dots & \dots \\ h_{k-1} & h_{k-2} & \dots & h_0 \end{bmatrix},$$

where $h_i = h(i)$ – impulse response value at time i.

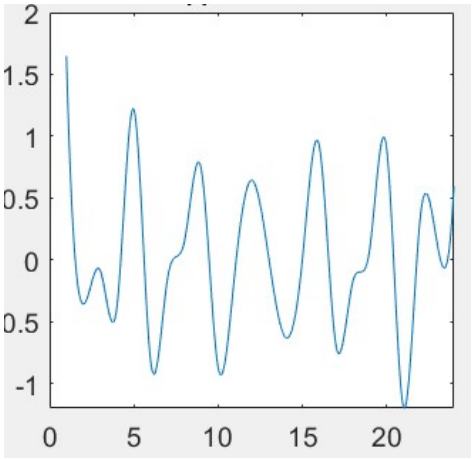
Let's give an example. Let the original signal be given as $sin(3t+5)+sin(8t),\ t=[1,24]$. And the impulse response $h(t)=\frac{1}{RC}e^{-t/RC}$, where R and C – the chain parameters.



a) Input signal. b) Impulse response

Fugire 4.1. — Graphs of the original signal and impulse response

Then the graph of the incoming signal:



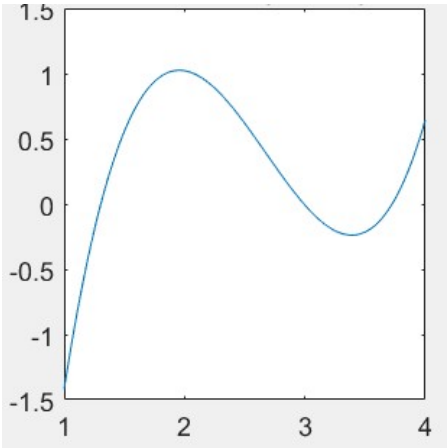
Fugire 4.2. — Incoming signal graph

We will receive the incoming signal in 4 stages: $t_1 = [1,4]$, $t_2 = (4,8]$, $t_3 = (8,20]$, $t_4 = (20,24]$. Recovery at the last stage comes down to finding a pseudo-inverse for the block $H=\left[h_4(4)h_3(3)h_2(2)h_1(1)\right]$.

When receiving the first part of the signal t_1 = [1,4], we find part of the reconstructed vector as $x_1 = h_1^+(1)y_1$:

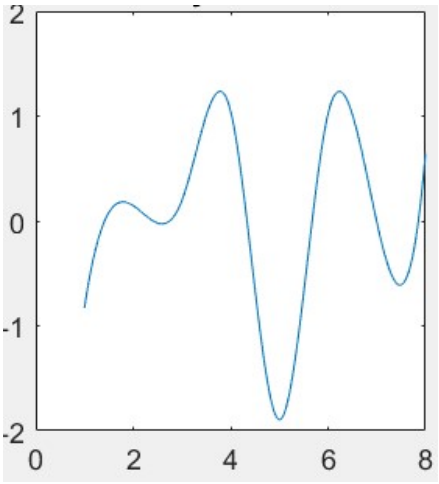
When the second part of the signal arrives y_2 source signal vector $\left[x_1^T x_2^T\right]^T$ is restored by the formula

$$\begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} E & h_1^+ h_2 \\ h_2^+ h_1 & E \end{bmatrix}^{-1} \begin{bmatrix} y_1 \\ y_2 \end{bmatrix}$$



Fugire 4.3. — The signal recovered at the first stage

Recovered signal graph $\left[x_1^T x_2^T\right]^T$

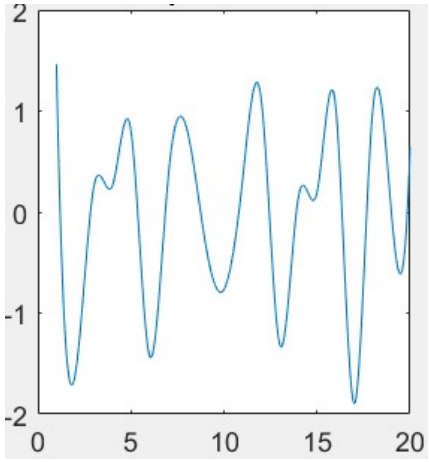


Fugire 4.4. — The signal recovered at the second stage

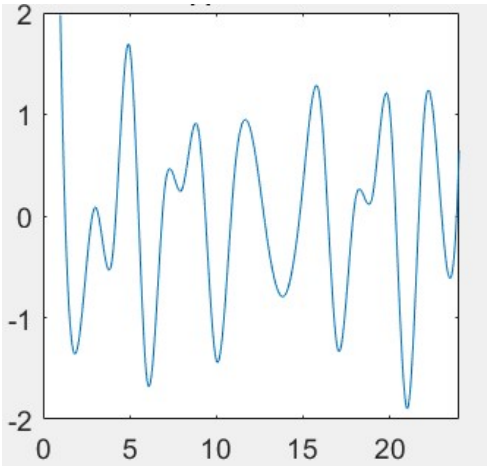
At the third stage we know $H_2^+ = \left[h_2 h_1\right]^+$.. Then we find the third part of the original signal using the formula

$$\begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} E & H_2^+ h_3 \\ h_3^+ H_1 & E \end{bmatrix}^{-1} \begin{bmatrix} y_1 \\ y_2 \\ y_3 \end{bmatrix}.$$

When receiving the last part of the signal, we find the pseudo-inverse matrix to the full matrix H, therefore, we obtain a completely reconstructed signal.



Fugire 4.5. — Signal recovered at the third stage



Fugire 4.6 — The signal completely restored at the fourth stage

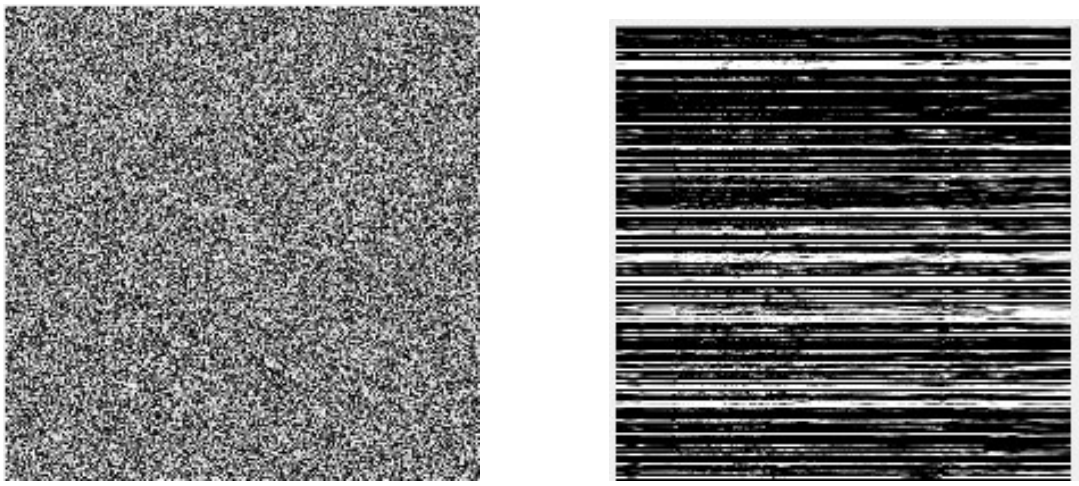
It can be seen that the reconstructed signal is completely identical to the original one. Obviously, the effectiveness of this algorithm depends on the size of the blocks, the conditionality of the matrix, and is most effective for large (wide) blocks.

4.3. Image recovery

Image restoration is similar to signal restoration, with the only difference being that in equation (4.6) X and Y are matrices. For example, consider an image of size 512×512 (see Fugire 4.7) [56; 57]. Let the original image be affected by a noise matrix $H_{512\times512}$ (see. The Fugire 4.8a) (based on normal distribution) as $Y = HX$. Under the influence of noise, the original image will take the form shown in the figure 4.8b.

Let us now assume that we receive the first data block of size 512×280, from which, using formula (4.2), we can obtain a partially restored image area of size 280×280 (Figure 4.9).

Figure 4.7. — Original image



a) Noise matrix b) Noisy image

Figure 4.8. — Image changes due to noise

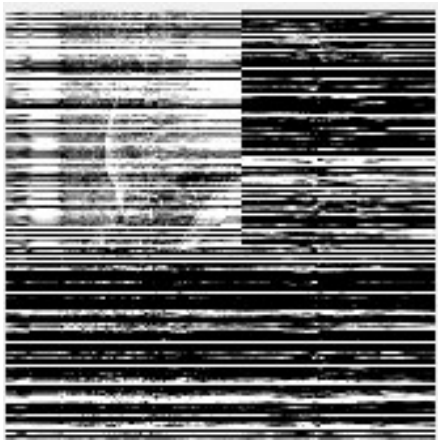


Figure 4.9. — Recovering from the first data block image

Suppose in the second step a block of size 512x120 arrives, from where we can get more information about the original image (the reconstructed image will have a size of (280 + 120) × (280 + 120)).

Figure 4.10. — Recovery in two blocks

Now let's take the last data block of size 512x120. As a result of the algorithm, the total size of all blocks will be 512×512, and we will receive a completely reconstructed image (Fig. 4.11).

Figure 4.11. — Restored image

In the case when the processing of data that can be reduced to a system of linear equations occurs sequentially (with sequential addition of data), the proposed algorithm based on the formula for block pseudo-inversion of matrices of full column rank shows a computational advantage compared to analogues, in addition, the algorithm copes well with the case of poor conditioned matrices, which is often encountered in practice.

4.4. Using block pseudo-inversion in neural network models

Pseudo-inversion methods are used to train neural networks; in some cases, pseudo-inversion can replace the back propagation method (see [50, 58-60]). The use of pseudo-inversion in training neural networks is quite natural, since it allows you to adjust the weight matrix so as to minimize the deviation of the network output from the training vector. Thus, a neural network without a hidden layer is nothing more than a regression equation, the coefficients of which can be found using pseudo-inversion.

The pseudo-inversion apparatus is widely used in the extreme learning method (ELM) (see [58-68]). The structure of networks used in extreme learning resembles the structure of neural networks based on radial basis functions (RBF) ([69-76]). Neural

networks of ELM and RBF methods have an input layer, one hidden layer and an output layer. An example of such a network is shown in Figure 4.12.

A feature of these networks is the linearity of the output layer, that is, the neurons of the output layer do not contain an activation function. While hidden layer neurons have an activation function. Initially, the weights of the hidden and output layers are set by random values, which means that we can neglect the settings of the weights of this layer and reduce the problem to setting the weights of the output layer, that is, solve the linear regression problem for the output layer. Let's consider the algorithm for training such a neural network.

Let $X_T = x_1, x_2, ..., x_T$ – set of incoming vectors, $Y_T = y_1, y_2, ..., y_T$ – set of training vectors. Then the output of a neural network with the structure shown in Figure 4.12 is calculated by the formula:

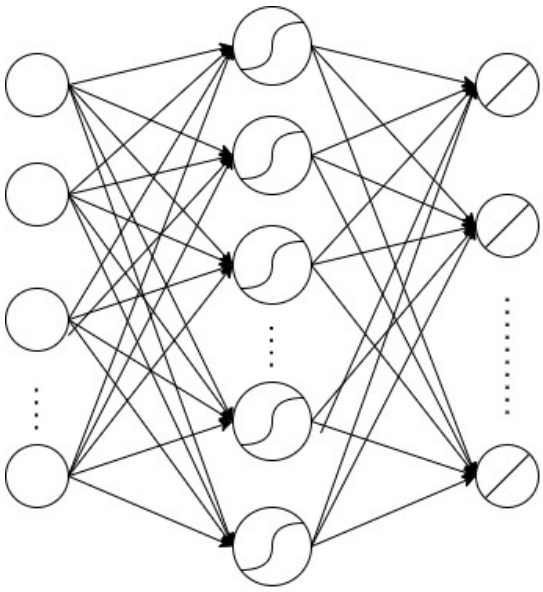


Figure 4.12. — Structure of RBF and ELM networks

$$y_{n,(t)} = \sum_{j=1}^{M} \omega_{nj}^{out} g\left(\sum_{i=1}^{d} \omega_{ji}^{hid} x_{i,(t)}\right), \tag{4.7}$$

where $y_{(t)}$ – network output corresponding to input $x_{(t)}$, and each output element y_n is the linear sum of M neurons in the hidden layer, weighted by the set ω_{ji}^{hid}. n is the index of the output neuron, j is the index of the hidden layer neuron, i is the

index of the input vector, *g()* is the activation function. Then, given the set of outputs of the hidden layer:

$$a_{j,(t)} = g(\sum_{i=1}^{d} \omega_{ji}^{hid} x_{i,(t)}),$$

we can form a matrix $A = \left[a_0 a_1 ... a_k\right]^T$, where each line is the output of the hidden layer for the *i*-th element from the training set and the corresponding matrix $Y = \left[y_0 y_1 ... y_k\right]^T$, where each line is the expected output value for the *i*-th element of the training set. In this notation, the task of training a neural network comes down to adjusting the weights of the output layer or, in other words, we need to select the weights of the output layer w^{out}, which will give the minimum error for the equation:

$$Aw^{out} = Y.$$

This problem can be solved using the pseudo-inverse Moore-Penrose matrix:

$$w^{out} = YA^{+}. \tag{4.8}$$

This approach has a number of significant advantages over other teaching methods. Firstly, we cannot get to a local minimum, since there is no need to look for a derivative and move along the gradient. Secondly, it is possible to obtain good model accuracy from a relatively small sample. Thirdly, for simple models this approach has a computational advantage compared to backpropagation.

Let's consider an example of training a neural network using the presented algorithm using simple functions as an example. As a training sample, we take the set t_i, $i = 1,...,20$, $t_i \in [-2,2]$ and the function values corresponding to this set $f(t) = \sin(t) + 5$. Let's choose the following network structure: one input neuron, three hidden layer neurons and one output neuron (see Figure (4.13)).

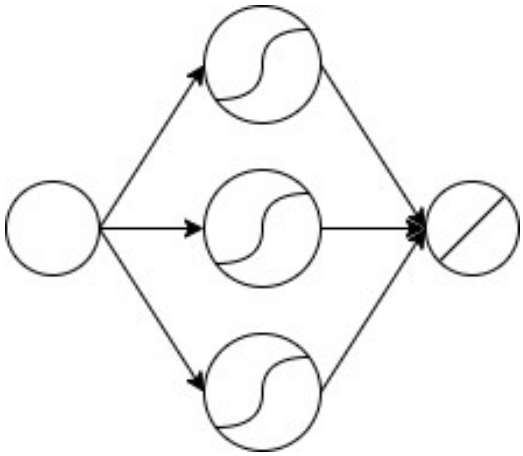


Figure 4.13. — Network structure for simple function prediction

We will use the sigmoid function as the activation function for the hidden word: $\sigma(x) = \frac{1}{1 + exp^{-x}}$. The result of training this network is shown in Fig. 4.14 (triangles mark predictions for the test data set).

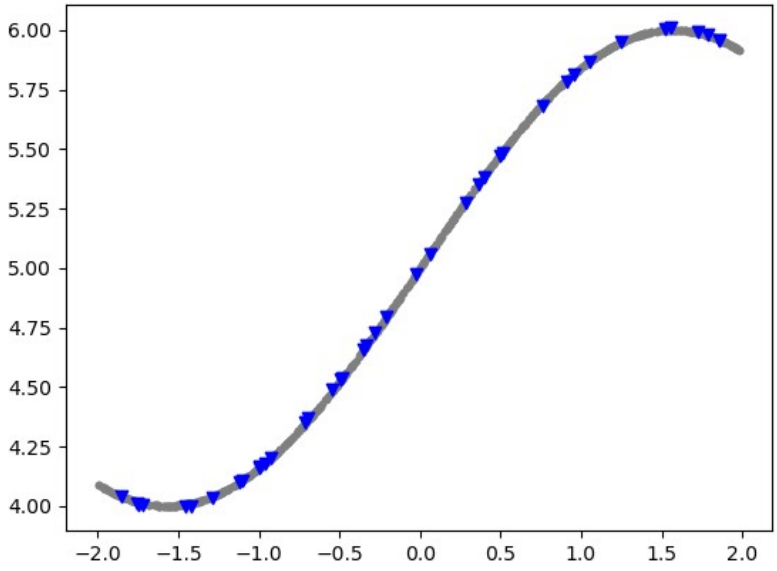


Figure 4.14. — The result of the network using the example of a simple function

This simple example shows that networks with elm and rbf structure are capable of producing good model accuracy on a small data set and a small number of hidden neurons.

4.5. Application of block pseudo-inversion in the problem of image recognition

However, the networks of the structure under consideration are also applicable for more complex problems. Let us consider as an example the benchmark task of recognizing handwritten digits (see [77]). Let's take a database as a data set MNIST [78].

Now let's set the structure of the neural network. The input will be a vector of length 784 (an image matrix of size 28×28 is written in one-dimensional form). The inner layer will consist of 7840 neurons (28 × 28 × 10) and the output layer will consist of 10 neurons. The index of the output neuron with the maximum value will be equal to the recognized digit. A sample of the input data is shown in Figure 4.15.

With a training set of 60,000 images, the accuracy of recognition training was about 97%. The program code in python can be viewed on github (see [79]).

Now let’s analyze how the number of neurons in the hidden layer affects the accuracy of training. In order to reduce calculations, we reduce the original data set to 1000. This dependence is presented in Fig. (4.16).

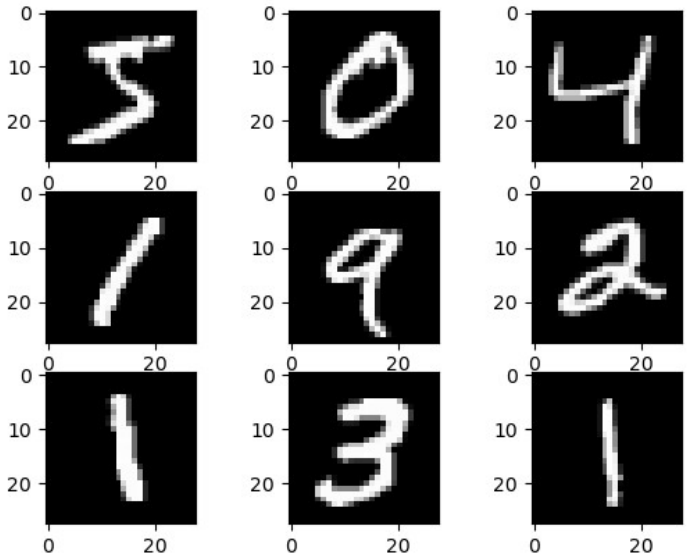


Figure 4.15. — The result of the network using the example of a simple function

Since one of the advantages of using pseudoinversion in training is the ability to obtain high accuracy of the model on a relatively small set of training data, we should separately consider the dependence of accuracy on the number of training samples.

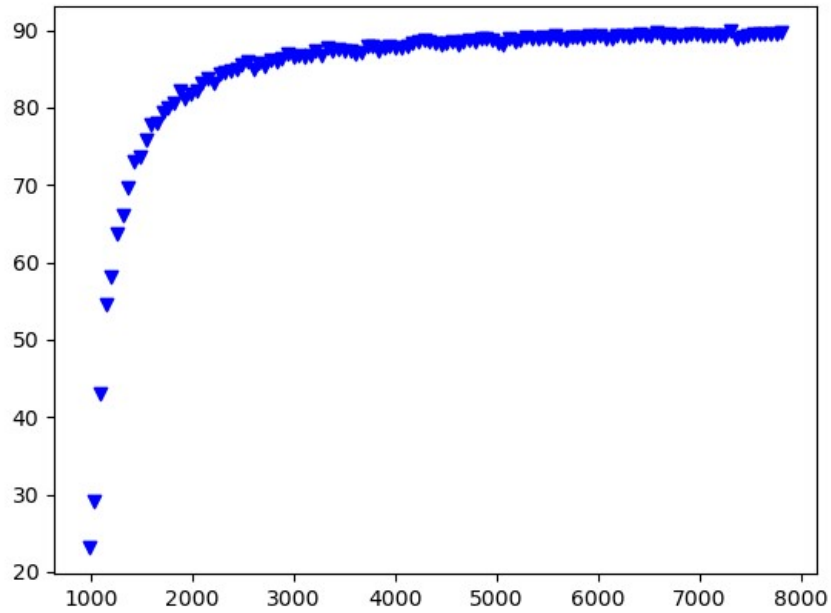


Figure 4.16. — Dependence of network accuracy on the number of neurons in the hidden layer

Figure 4.17 shows this dependence with 1500 hidden neurons. As you can see, changing the size of the training sample (several times) does not lead to a significant change in the accuracy of the model (fluctuations are no more than 5%). Consequently, in tasks where such an error in accuracy is acceptable, you can significantly reduce the cost of collecting and storing training data, and reduce training time. It should be remembered that it is quite difficult to formulate a network structure in advance that will provide the necessary accuracy of the model, and in practice, often, network parameters are selected based on experiments with various variations of model parameters.

It can be noted that since in the internal layer of the network in Figure (4.12) the weights are chosen arbitrarily and pass through a nonlinear formula, the output matrix of the weights in formula (4.8) with a high degree of probability will be of full column

rank. This means that in this case the block pseudoinversion formula (2.22) is applicable.

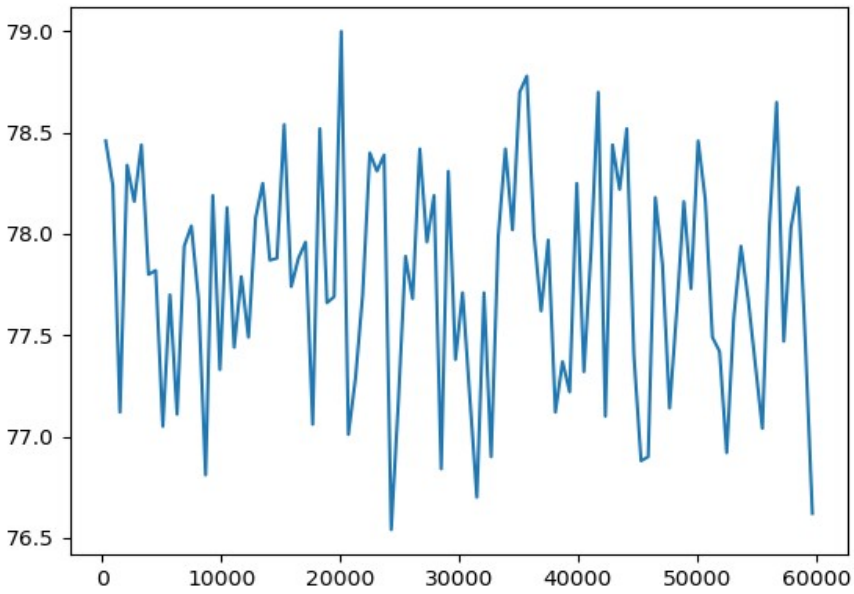


Figure 4.17. — Dependence of network accuracy on the size of the training sample

This formula can be used to increase the efficiency of calculating a pseudoinverse matrix, namely, parallelizing its calculation. To do this, we will split the original matrix A into 2 blocks, then split each block into 2 more: $A = \begin{bmatrix} [A_1 A_2] \\ [A_3 A_4] \end{bmatrix}$. Next, using the formula in different threads, we will find pseudo-inverses to each of the blocks $A^+_{1-2} = [A_1 A_2]^+$, $A^+_{3-4} = [A_3 A_4]^+$, and, knowing the pseudoinverses to the blocks, using formula (2.22) we find the pseudoinverse to the entire matrix. The code for this algorithm can be viewed on github (see [79]).

This parallelization is quite crude, in the sense that we did not use standard approaches for this area for parallelizing calculations (see [80-93]), which means that the influence of the operating system, interaction with memory, caches, and the costs of creating and running additional streams. However, the influence of these effects should decrease as the volume of calculated data increases, in our case: the larger the training sample and the number of hidden layer neurons, the more noticeable should be the difference in training time between a dual-threaded and single-threaded algorithm. This is confirmed by graph (4.18). This graph shows the dependence of

training time on the number of hidden layer neurons (training sample size 60,000) for single-threaded and double-threaded algorithms. It can be seen that, starting from a certain size, the training time of a single-threaded program begins to exceed the training time of a double-threaded program.

Obviously, with a further increase in the number of internal layers, the difference in training time will only increase. With a number of five thousand neurons in the hidden layer, the difference in learning was 25%, with seven thousand – already 35%.

4.6. Using the block pseudo-inversion formula in the problem of predicting the optimal markup

Let's consider an example of using the block pseudo-inversion formula in the problem of predicting the optimal markup for goods of a retail chain, having a three-year sales history, solved in the company Surf Studio.

Predicting the markup in the general case is a complex task, to solve which one has to resort to various types of heuristics. The markup is based on many factors that can be divided into two categories: controllable and uncontrollable. Uncontrollable factors include such factors as: inflation, commodity prices, exchange rates and even the weather. Unfortunately, due to commercial secrets, we cannot provide algorithms for collecting and taking into account the influence of uncontrollable factors, as well as some controlled parameters of the model. However, even with a limited set of parameters, we will be able to form a simplified mathematical model suitable for further improvement, taking into account the specifics of the work of a particular company.

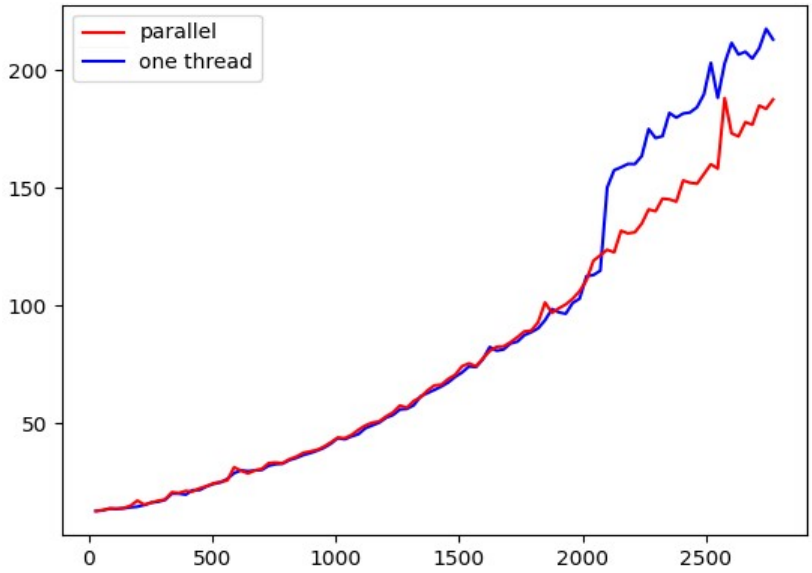


Figure 4.18 — Execution time of a single-threaded and double-threaded program

To form a predictive model, we have the following data: how much of the product was sold, how much is in stock, retail price, purchase price, profit received. The data is broken down weekly and for different regions. A sample of the available data is shown in the figure. 4.19.

sold in stock purchase price retail price

sold	in stock	purchase price	retail price
57	122	1299.00	2100.0
78	80	1352.22	2199.0
62	101	1277.53	2299.0
75	106	1299.0	2100.0
96	92	1299.0	2100.0
3754	8882	85.45	144.99
3566	4234	79.50	125.0

Figure 4.19. — Sample data for forming a predictive model

Based on data analysis and some heuristic indicators, Surf compiled the following mathematical model:

$$\left[X_1 E\left[A_m\right]\right] P_{pr} = M, \tag{4.9}$$

where X_1 – how many units of the product were sold, A_m – how much product is left in stock, $E[A_m]$ – how many goods from the warehouse can be expected to be sold (mathematical expectation), P_Z – purchase price, P_{pr} – required quantity: price prediction, M – profit received. From we can get the required value explicitly:

$$P_{pr} = \left[X_1 E\left[A_m\right]\right]^{+} M, \tag{4.10}$$

Since data arrives over time and in large volumes, this model must be constantly recalculated, and here, to reduce calculations, the block pseudo-inversion formula (2.22) is applicable.

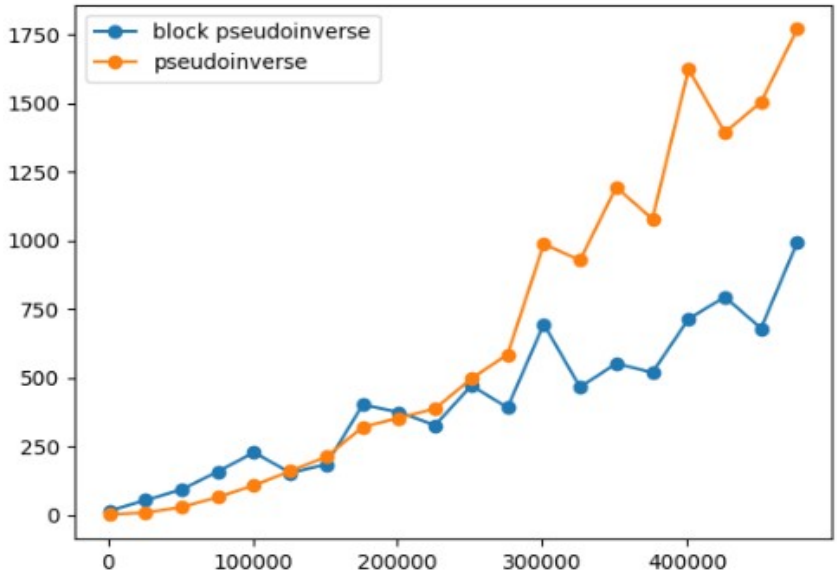


Figure 4.20 — Model recalculation time

Figure 4.20 shows a graph of model recalculation with the arrival of new data. The horizontal axis shows the amount of received data, and the vertical axis shows the model calculation time (in minutes). It can be seen that the more data, the more significant the time difference becomes, while the accuracy of the calculation is maintained. In this case, the only drawback is the need to store the pseudo-inverse matrix on the server. But in this case, this is a small cost: storing an additional matrix takes about a gigabyte, which, given the large set of source data, is an insignificant factor.