Implementation Aspects of a Single-Layer LSBDL Model

Alireza Amiri-Simkooei[1*], Farideh Sabzehee[2], Mirjam Snellen[1]

[1] Dept. of Control and Operations, Delft University of Technology, 2629 HS Delft, the Netherlands - a.amirisimkooei@tudelft.nl, m.snellen@tudelft.nl

[2] Dept. of Geomatics Engineering, University of Isfahan, Isfahan, Iran - sabzfarideh@gmail.com

[*] Corresponding author

Keywords: Least-squares-based deep learning, Least-squares method, Deep learning, Single-layer model, Surface fitting problem, Steepest descent method.

Abstract:

This paper presents the implementation of the single-layer least-squares-based deep learning (LSBDL) model, optimized using the steepest descent method. As a showcase, the work numerically validates LSBDL's performance in complex non-linear applications, such as surface fitting. LSBDL is proposed as a transparent deep learning solution, uniquely merging the theoretical robustness and quality control capabilities of the least squares (LS) method with the flexibility of deep learning (DL) models. Unlike conventional black-box DL architectures, the LSBDL framework naturally provides statistical quality assessment metrics, including the covariance matrix of estimated parameters and precision of predicted outcomes. This enables seamless model mis-specification and outlier detection using established reliability theory. The key focus of this study is the model's demonstrated efficiency, accuracy, and performance in complex non-linear applications. In a complex surface fitting application, the implemented LSBDL model achieved a root mean square error (RMSE) of 0.0021, which is significantly lower than the simulated noise level. Furthermore, the estimated LS residuals are consistent with the simulated (and also estimated) standard deviation of $\sigma = 0.01$. The implemented model offers an effective, statistically grounded, and numerically efficient solution for handling complex non-linear problems, particularly those involving heterogeneous and correlated observations. All hyperparameters, initialization steps, optimization, and validation procedures are thoroughly discussed. The Matlab and Python code is freely available at: https://github.com/tud-dasaa/lsbdl.v1.

1. Introduction

The 21st century is characterized by rapid increase of big data, offering unprecedented opportunities while also posing serious challenges for its effective processing and meaningful interpretation (Williams, 2017). Within the realm of artificial intelligence (AI), machine learning (ML) has emerged as a pivotal technology, influencing a wide range of domains. The inherent complexity of many ML algorithms necessitates the exploration of novel approaches for data analysis and knowledge extraction (Jordan and Mitchell, 2015). Deep learning (DL), a branch of ML, trains complex models and has achieved notable success across many applications, drawing strong research interest (Emmert-Streib et al., 2020). This success, however, is often shadowed by the "black-box" nature of deep neural networks, which hinders interpretability and poses challenges in areas such as outlier detection and robust statistical inference (Najafabadi et al., 2015). To address this lack of transparency, the development of eXplainable AI (XAI) has emerged as a prominent and active research area (Arrieta et al., 2020). In response to these challenges and the growing demand for transparent and efficient learning paradigms, a novel deep learning method, least-squares-based deep learning (LSBDL), has recently been introduced (Amiri-Simkooei et al., 2024). LSBDL combines the interpretability of least squares theory with the flexibility of deep learning, offering a promising method to complex computational tasks from an XAI perspective. A conceptual overview of data science techniques is shown in Fig. 1, which illustrates the hierarchical relationship between AI, ML, DL, and artificial neural networks (ANNs) (Choi et al., 2020). ANNs, inspired by biological neural systems (Haykin, 1994), are the core computational models in DL. They are particularly effective for problems involving large datasets and nonlinear relationships, where traditional methods may fail (Bishop, 1995). An ANN consists of interconnected layers of artificial neurons — input, hidden, and output — where weighted inputs are passed through activation functions to produce outputs. Key features of ANNs include learning from data, generalization, robustness, and the ability to approximate complex relationships (Sabzehee et al., 2018). Common architectures, such as multi-layer feed-forward networks (FFN), are widely used in regression and classification tasks (Sabzehee et al., 2023). Model structure, e.g. number of layers and neurons, is typically determined through experimentation (McKinnell, 2008). Within this context, LSBDL positions itself as a novel approach that retains the modeling flexibility of ANNs while enhancing interpretability through its grounding in least squares theory. This paper further explores the formulation of deep learning (DL) within the framework of standard least squares theory, with a focus on practical implementation. We describe how LS-

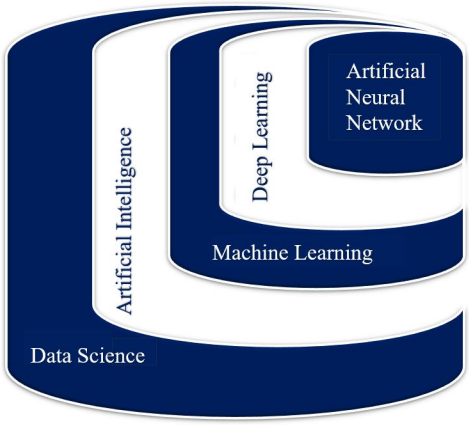


Figure 1. Subfields of data science.

BDL constructs a design matrix by convolving a feature (input) matrix with an unknown weight matrix, followed by a nonlinear activation function. A single-layer LSBDL model is trained using the steepest descent (SD) optimization method. This approach allows for the direct application of least squares theories, such as prediction error analysis and hypothesis testing (Teunissen, 2000, 2018), while enhancing the transparency and interpretability of DL models (Amiri-Simkooei et al., 2024). The remainder of this paper is structured as follows: Section 2 reviews and presents the rationale behind the development of least-squares-based deep learning (LSBDL) and discusses its inherent transparency, detailing the training process of the design matrix using the steepest descent (SD) method. Section 3 focuses on the implementation of LSBDL and illustrates its performance through a surface fitting problems. Finally, Section 4 concludes the paper by summarizing the key findings and outlining potential avenues for future research.

2. Single-layer LSBDL model

2.1 Background

This section reviews the least-squares-based deep learning (LSBDL) model (Amiri-Simkooei et al., 2024), proposed in the context of a multivariate linear model. The multivariate model, relates r observation vectors (organized as an $m \times r$ matrix $Y = [\mathbf{y}_1, \ldots, \mathbf{y}_r]$) to r unknown vectors (organized as an $n \times r$ matrix $X = [\mathbf{x}_1, \ldots, \mathbf{x}_r]$); in LSBDL, r refers to the number of output variables. A multivariate linear model is of the form (Amiri-Simkooei, 2009):

$$Y = AX + E \tag{1}$$

where the same $m \times n$ design matrix A connects y_i to x_i for all $i = 1, \ldots, r$, and $E = [e_1, \ldots, e_r]$ is an $m \times r$ residual matrix. The covariance matrix of Y can be expressed as a Kronecker product $\otimes$ of two matrices as:

$$\mathsf{D}(Y) = Q_{\mathrm{vec}(Y)} = \Sigma \otimes Q \tag{2}$$

where $\mathrm{vec}(.)$, the vector operator, converts a matrix into a column vector, the $r \times r$ matrix Σ expresses covariances among different output (classes), and the $m \times m$ matrix Q expresses covariances among different observations within a class. The least squares estimate of the unknown parameters is

$$\hat{X} = (A^T Q^{-1} A)^{-1} A^T Q^{-1} Y \tag{3}$$

The least squares estimates of the observation and residual matrices are

$$\hat{Y} = A\hat{X}, \text{ and } \hat{E} = Y - \hat{Y} \tag{4}$$

A special case of the multivariate model, is the univariate model, which relates a single observation vector $\mathbf{y}$ to a single unknown vector $\mathbf{x}$ (i.e. $r = 1$). Classical multivariate linear models assume that both the design matrix and observation matrix are known. In LSBDL, the design matrix is unknown and must be determined through a supervised learning process. This process constructs the $m \times n$ design matrix A from an $m \times k$ feature matrix D (input), where each row of D contributes to make one row in the design matrix. If a row of D is denoted as $\mathbf{d} = [d_1, d_2, \ldots, d_k]^T$, we introduce the unknown weights $\mathbf{w} = [w_1, w_2, \ldots, w_k]$ and form their linear combination as $\mathbf{d}^T\mathbf{w} = \sum_i d_i w_i$. For the sake of brevity, we assume $d_k = 1$, so the last term in the linear combination becomes $d_k w_k = w_k = b$, where b is a bias term. In deep learning applications, these linear combinations of input variables are typically followed by activation functions that introduce nonlinearity. The output of a single neuron is therefore given by $y = \mathsf{a}(\mathbf{d}^T\mathbf{w})$, where $\mathsf{a}(.)$ is an activation function. Figure 2 illustrates a typical example of an activation function, i.e. 'sigmoid' and its derivative, which play an important role in gradient-based optimization methods.

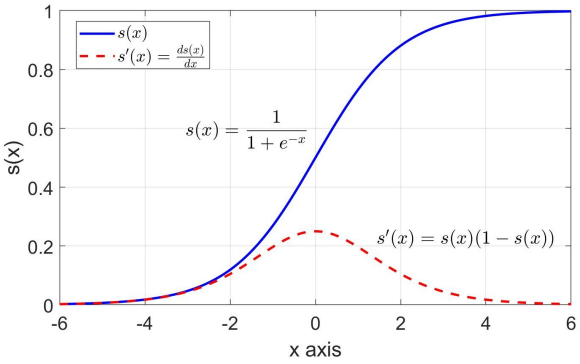


Figure 2. Sigmoid activation function and its derivative.

A single neuron can only model simple relationships and is usually insufficient to capture complex data patterns. To model more complex relationships, we introduce multiple neurons in the same layer, each with its own weight vector $\mathbf{w}_j$. The outputs of these neurons are then combined in a subsequent (output) layer. In this formulation, let $x_1, \ldots, x_n$ denote the weights connecting the outputs of the n neurons to the output layer, which we assume applies an identity activation function. The resulting model output can then be written as:

$$y = f(\mathbf{d}) = \mathsf{a}(\mathbf{d}^T\mathbf{w}_1)x_1 + \cdots + \mathsf{a}(\mathbf{d}^T\mathbf{w}_n)x_n \tag{5}$$

This composition of linear combinations and nonlinear activation functions enables neural networks to capture hidden patterns in data and is fundamental to data-driven machine learning. Together with the *universal approximation theorem*, equation (5) states that any continuous real-valued function of k real variables can be uniformly approximated with arbitrary precision using finite linear combinations (here n) of suitable activation functions, see Cybenko (1989); Costarelli et al. (2013). So far, we considered the case of a single output ($r = 1$). However, many practical problems require multiple outputs, such as in multi-class classification or multi-output regression. To handle this, we assign a separate set of weights for each output unit. Let the total number of outputs be r, and denote the output vector as $\mathbf{y} = [y_1, \ldots, y_r]^T$. Then, for each output y_j, we introduce a distinct set of weights $x_{1j}, \ldots, x_{nj}$ connecting the n hidden neurons to the j-th output. The j-th output is computed as:

$$y_j = f_j(\mathbf{d}) = x_{1j}\mathsf{a}(\mathbf{d}^T\mathbf{w}_1) + \cdots + x_{nj}\mathsf{a}(\mathbf{d}^T\mathbf{w}_n) \tag{6}$$

where $j = 1, ..., r$. The above formulation naturally extends the single-output case to multiple outputs and reflects the structure used in the fully connected layers of LSBDL (Amiri-Simkooei et al., 2024). Let m denote the number of observations and k the number of features (i.e., each observation has k features). We define the feature matrix as $D \in R^{m \times k}$. The matrix D is used to construct the design matrix via the transformation

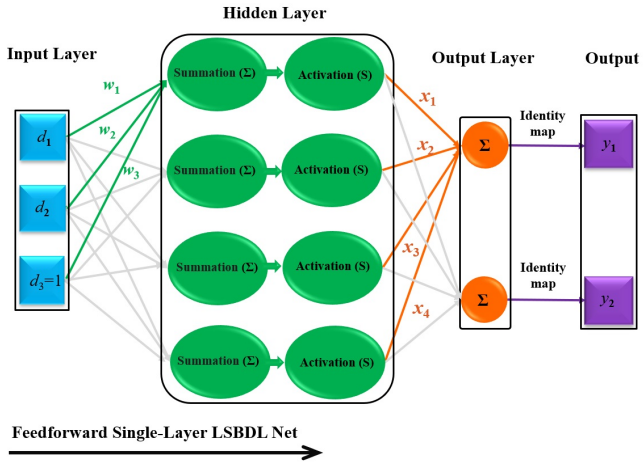


Figure 3. Architecture of a single-layer LSBDL network.

$A = \mathsf{A}(DW)$, which results in a matrix-form expression of Eq. (6):

$$Y = \mathsf{A}(DW)X + E, \quad Q_{\mathrm{vec}(Y)} = \Sigma \otimes Q \tag{7}$$

where $A(.)$ is an activation function applied elementwise to the entries of DW. This system of equations is nonlinear with respect to W; however, once this matrix is fixed, the system becomes linear in X. Figure 3 illustrates a feedforward architecture of an LSBDL network with $k = 3$ input variables (features $\mathbf{d} = [d_1, d_2, d_3 = 1]^T$), $n = 4$ neurons in the hidden layer, and $r = 2$ output variables (y_1 and y_2). The weights associated with the first hidden neuron are w_1, w_2, and w_3. The weighted sums of the inputs are passed through an activation function to produce the hidden layer outputs. These outputs are then fed into the output layer, where they are combined using weights x_1, x_2, x_3, and x_4. Finally, the output layer applies an activation function (the identity map in this case) to produce the final output. The matrix representation of this example can be explicitly written as: a matrix-form expression of Eq. (6):

$$\mathsf{A}\left(\begin{bmatrix} : & : & : \\ d_{i1} & d_{i2} & d_{i3} \\ : & : & : \end{bmatrix}\begin{bmatrix} w_{11} & w_{12} & w_{13} & w_{14} \\ w_{21} & w_{22} & w_{23} & w_{24} \\ w_{31} & w_{32} & w_{33} & w_{34} \end{bmatrix}\right)\begin{bmatrix} x_{11} & x_{12} \\ x_{21} & x_{22} \\ x_{31} & x_{32} \\ x_{41} & x_{42} \end{bmatrix} = \begin{bmatrix} : & : \\ y_{i1} & y_{i2} \\ : & : \end{bmatrix} \tag{8}$$

where i ranges from 1 to m, and the equation is indeed in the form of Eq. (7).

2.2 Optimization of LSBDL Model

Given matrices Y and D, the supervised learning leads to the following least squares criterion: The design matrix $A = \mathsf{A}(D\,W)$ is unknown and needs to be trained/estimated using the least squares approach.

$$(\hat{W}, \hat{X}) = \arg\min_{(W,X)} \frac{1}{2}\|Y - \mathsf{A}(D\,W)X\|^2 \tag{9}$$

where $\Sigma \otimes Q$ is used to weigh $\|.\|^2$. We can solve for the following minimization issue $\|E\|^2 = \|Y - AX\|^2 \rightarrow \min$ if we assume that $A = \mathsf{A}(D\,W)$ is supplied. As a result, we estimate X and W in two steps. It is known that $\hat{X} = (A^T Q^{-1} A)^{-1} A^T Q^{-1} Y$ is the global minimizer for X.

$$W^{(t+1)} = W^{(t)} + \alpha\, D^T(Q^{-1}\hat{E}\Sigma^{-1}\hat{X}^T \odot A') \tag{10}$$

Algorithm 1: Implementation of LSBDL using SD method

Input:
1. obtain data/feature matrix D
2. observation matrix Y ($\Sigma = I_r$ and $Q = I_m$)
3. initialize weight matrix $W^{(0)}$ and $\Delta W^{(0)} = 0$
4. set regularization parameter κ
5. set learning rate parameter α
6. set momentum parameter μ
7. set softening parameter $s \in (0, 1)$
8. set maximum number of iterations ($t_{\max}$)

begin
do for $t = 0$ till convergence
compute the activation matrix A and its derivative A'
obtain the least squares estimates $\hat{X} = (A^T A + \kappa I_n)^{-1} A^T Y$
compute the least squares residuals $\hat{E} = Y - A\hat{X}$
compute the gradient matrix $\nabla\phi = -D^T(\hat{E}\hat{X}^T \odot A')$
soften the gradient by transformation $\nabla\phi = \mathrm{sgn}(\nabla\phi) \odot |\nabla\phi|^s$
compute weights' corrections $\Delta W^{(t+1)} = -\alpha\,\nabla\phi + \mu\,\Delta W^{(t)}$
update the weights $W^{(t+1)} = W^{(t)} + \Delta W^{(t+1)}$
increase the counter $t = t + 1$
end do
while $t < t_{\max}$ **repeat** t
end

Figure 4. Symbolic algorithm for implementation of a single layer LSBDL model formulated by the steepest descent method (Amiri-Simkooei et al., 2024, after).

This improves W in the SD algorithm through iterations. It includes the matrices D and A', the observation covariance matrix $Q_{\mathrm{vec}(Y)}$'s matrices Σ and Q, and the least squares estimations of $\hat{E}$ and $\hat{X}$. To carry out the previously mentioned SD approach, the procedure starts with an appropriate starting weight $W^{(0)}$ and iterates through the previously mentioned equations to modify the weights W. When the objective function $\phi(W)$ has sufficiently decreased or when $\|W^{(t+1)} - W^{(t)}\|$ drops below a predetermined threshold ϵ (Teunissen, 1990), for example, the iterations will continue until the convergence. Alternatively, the process may be terminated after a fixed number of iterations, depending on performance on the testing data. Once the weights (and biases) have been determined, they are used to compute the activation (design) matrix A. A summary of these processes is provided in Algorithm 1 (Fig. 4).

LSBDL combines least squares theory and deep learning to create a method with the following key characteristics: improved interpretability and transparency; intrinsic quality control; robust error and outlier detection and removal using reliability and hypothesis testing; the ability to embed prior knowledge and physical laws; utilization of classical least squares tools; and versatility across scientific and engineering applications (Amiri-Simkooei et al., 2024).

2.3 Hyperparameters and Their Roles

- **Activation function**: An activation function introduces nonlinearity to allow for complex modeling. For the example considered (see the next section), the relation between $[\mathbf{1},\ u,\ v]$ and $f(u, v)$ is not linear, which requires introducing nonlinearity using activation functions such as `sigmoid`, `tanh` or `softplus`.

- **Number of neurons (n = 5)**: Number of neurons determines the width of the hidden layer, thus controlling the

model's capacity. A small value of n can lead to underfitting, while a large n may cause overfitting. The choice of n also depends on the number of features k; in this example, $k = 3$ likely requires a larger n), whereas $k = 6$ likely needs a smaller n.

- **Softening parameter** ($s = 0.5$): This parameter controls the degree of nonlinearity applied to the gradient updates during training. To mitigate the risk of large gradients, which can cause saturation (where gradients vanish) and increase the risk of overfitting, we propose a gradient softening method. Instead of using the raw gradients, we apply a fractional exponent transformation using the $s \in (0, 1)$ softening parameter. A smaller s leads to stronger regularization by efficiently compressing large gradient values. We found $s = 0.5$ (i.e. taking the square root of the absolute gradient entries) to be an effective choice for preventing excessive weight growth.

- **Regularization parameter** ($\kappa = 10^{-6}$): In general, the regularization parameter prevents overfitting in DL models. For LSBDL, it stabilizes matrix inversion by penalizing large weights. We use the Tikhonov regularization method, which has been widely used with the least squares methods Tikhonov (1963).

- **Learning rate** ($\alpha = 0.02$): It determines the step size in the direction of the gradient. A (too) small $\alpha = 0.02$ can lead to slow convergence and may get stuck in local minima, whereas a (too) large learning rate causes the model to overshoot minima and even diverge. Depending on the application, typical values range from 10^{-10} to 1.

- **Momentum parameter** ($\mu = 0.9$): A momentum parameter is used in LSBDL to accelerate convergence and smooth updates by incorporating a fraction of past gradients. It typically ranges from 0 to 0.99; we use 0.9 as a common default. Lower values (e.g. 0.1) give rise to slower and more cautious learning, while higher values (e.g. 0.99) speed up convergence but may cause overshooting or instability if not properly tuned. Momentum helps LSBDL move more smoothly and stay on track, even when the loss function is bumpy or uneven.

- **Activation parameter** ($actp = 1$): In LSBDL, one hyperparameter is the activation function parameter $actp$. It controls the horizontal scaling of the activation function, which allows it to be stretched or compressed. For example, when using a sigmoid activation, larger $actp$'s make the curve steeper, so the output becomes saturated quickly (becomes close to 0 or 1 near the origin). In contrast, smaller $actp$'s flatten the curve, so the function stays away from saturation (i.e. 0 or 1) for a wider range of input values. This parameter was set to its default value, $actp = 1$.

- **Epochs (= 200)**: Total number of iterations of the optimization loop.

3. Implementation in Matlab and Python

As mentioned, LSBDL integrates the structure of linear models into deep learning via the least-squares principle. We now present an implementation case study using a synthetic nonlinear surface to illustrate how LSBDL learns hidden patterns and provides predictions with low residuals. The implementation is performed both in Matlab and Python, and is freely accessible at: https://github.com/tud-dasaa/lsbdl.v1. In this paper, we focus on explaining selected parts of the Matlab code implementation for the sake of brevity.

3.1 Problem Setup

The target surface is defined as the following mathematical formula:

$$y = f(u, v) = (u^2 + v^2)e^{-u^2-v^2}, \tag{11}$$

Figure 5 shows the original function evaluated on a regular grid over the intervals $[-2, 2]$ along both the u and v axes.

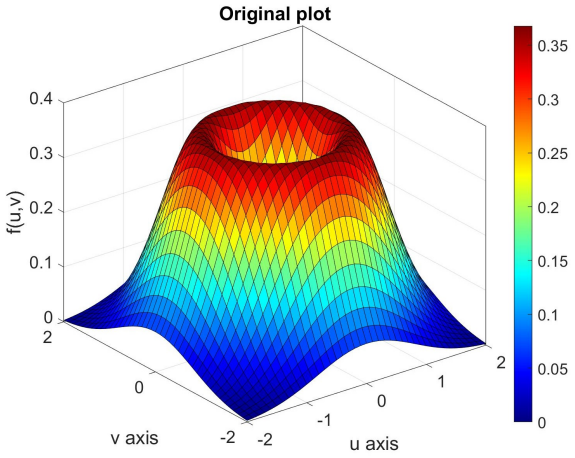


Figure 5. Function $y = f(u, v) = (u^2 + v^2)e^{-u^2-v^2}$.

Training and prediction are performed on this function over the domain $[-2, 2]\times[-2, 2]$. The training data consists of $m = 500$ points simulated using the above function, which is also corrupted with Gaussian noise. Figure 6 shows a typical example of such simulated data.

```
% simulate 500 training points
m = 500;
% generate u and v in [-2, 2] interval
u = 4*(rand(m,1)-0.5);
v = 4*(rand(m,1)-0.5);
% obtain training output data
y = (u.^2 + v.^2) .* exp(-u.^2 - v.^2);
% add normal random noise (std=0.01)
y = y + 0.01*randn(m,1);
Y = y;
```

For this example, two feature matrices have been considered:

- Case 1: $\mathbf{D} = [\mathbf{1},\ u,\ v]$
- Case 2: $\mathbf{D} = [\mathbf{1},\ u, v,\ uv,\ u^2,\ v^2]$

Case 1 considers the simplest case for which the features are just the point coordinates u_i and v_i, $i = 1, ..., m$ and the bias term (the column of 1), resulting an $m \times (k = 3)$ feature matrix D. Case 2 takes some of the nonlinearity by including more features u_iv_i, u_i^2, and v_i's. This results in an $m \times (k = 6)$ feature matrix D.

```
Case = 2; % Case = 1;
% construct the feature matrix
if Case==1
   D = [ones(m,1), u, v];
else
   D = [ones(m,1), u, v, u.*v, u.^2, v.^2];
end
```

This contribution has been peer-reviewed. The double-blind peer-review was conducted on the basis of the full paper.
https://doi.org/10.5194/isprs-annals-X-4-W8-2025-85-2026 | © Author(s) 2026. CC BY 4.0 License.

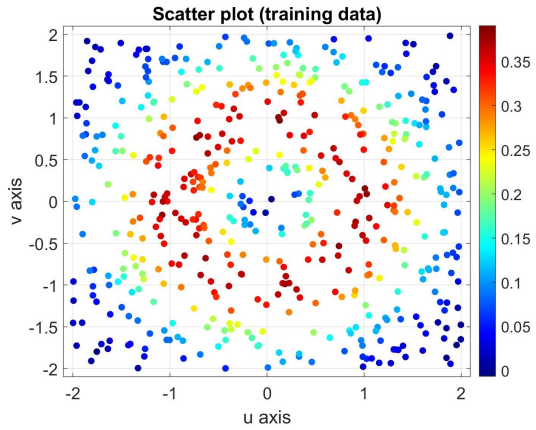
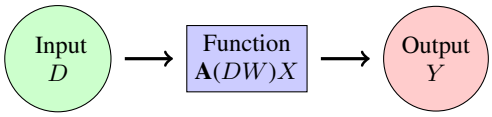


Figure 6. Scatter plot of $m = 500$ points from $y = f(u, v)$, with (u, v) sampled uniformly over the plane and y perturbed by Gaussian noise with a standard deviation of $\sigma = 0.01$.

The goal now is to establish the link between the input matrix D and the output matrix Y (for the example considered in this study, Y is a vector, indicating that it has only one output).



The function $\mathrm{A}(DW)X$ is estimated by training a single-layer LSBDL model using the SD method (Algorithm 1). The implementation employs a network with nonlinear activation functions. The weights are updated using gradient descent with some hyperparameters such as softening and momentum. The hyperparameters and their roles have already been explaind in Section 2.3. All hyperparameters were defined within a Matlab structure `P`, which makes them easy to manage, tune, and modify independently. The activation function, specified separately via the variable `Activation`, was set to `'sigmoid'` in this implementation.

```
% activation function
Activation = "sigmoid";
% number of observations
P.m = size(D,1);
% number of features
P.k = size(D,2);
% number of output variables
P.r = size(Y,2);
% number of unknowns (neurons)
P.n = 5; % tunable
% softening parameter
P.spar = 0.5;
% regularization parameter
P.kappa = 1e-6;
% learning rate parameter
P.alpha = 2e-2;
% momentum parameter
P.mu = 0.90;
% activation parameter
P.actp = 1;
% Number of iterations
P.Epochs = 200;
```

The code is then divided into four core functions:

- `activation_function`: Computes the design matrix A and its derivative A'.
- `lsbdl_sd_initialize`: Initializes weights, activations, and estimates the parameters.
- `lsbdl_sd_optimize`: Iteratively updates the weights using steepest descent.
- `lsbdl_sd_test`: Predicts outputs for test data using trained parameters.

Here are some detailed information. **Step 1:** The implementation of the LSBDL method follows a structured approach, where the hyperparameters (in structure `P`) and primary data (all matrices), organized within the structure `MAT`, are used. The initialization step involves setting the feature matrix `D` and the target matrix `Y`, and subsequently calling the function `lsbdl_sd_initialize` to set up the necessary components for the LSBDL algorithm.

```
% Initialization step
MAT.D = D;
MAT.Y = Y;
MAT = lsbdl_sd_initialize(MAT, P, Activation);
```

The function `lsbdl_sd_initialize` uses key variables such as the number of features `k`, neurons `n`, and the regularization parameter κ to initialize the weight matrix `W` with uniform random values, and then initialize `X`. To prevent the activation function from operating in its saturated regime, the weights are rescaled by a factor inversely proportional to the average standard deviation of the projected input data. The activation function is then applied to compute the design matrix `MAT.A` and its derivative `MAT.Ap`. Finally, the initial values for the least-squares solutions `MAT.X`, estimated output `MAT.Yh`, and residual error `MAT.E` are computed for use in subsequent iterations. The code for the function is provided in Appendix. **Step 2:** The core optimization routine in the LSBDL is executed iteratively across a user-defined number of epochs (here 200). This is implemented through a loop that repeatedly calls the `lsbdl_sd_optimize` function. In each iteration, the function updates the model parameters W and X to minimize the discrepancy between observed and modelled data, using a steepest descent (SD) method.

```
% Optimization step
for it = 1:P.Epochs
    P.it = it;
    MAT = lsbdl_sd_optimize(MAT, P, Activation);
end
% estimate the covariance matrix Sigma
Sigma = MAT.Sigma;
% standard deviation of LS fit
EstimatedStd = MAT.Std(end);
fprintf('Estimated standard deviation: %.4f\n',
    EstimatedStd)
```

This function first computes the gradient of the loss function with respect to the weight matrix W, which incorporates the derivative of the chosen activation function. This gradient is then softened using a power-based regularization controlled by a softening parameter $spar$. The weight update combines this softened gradient with a momentum term and a learning rate (all tuned to their optimized values). Once the weights are updated, the design matrix A and its derivative are recalculated, and the updated system is solved using a regularized least-squares formulation to re-estimate the unknown parameters X. The function then recomputes the estimated observations $\hat{Y} = AX$, and consequently estimates the residuals $E = Y - \hat{Y}$. , and stores the standard deviation of these residuals at each iteration. After all epochs are completed, the final residual matrix is used to compute the estimated covariance matrix and standard deviation of the fit:

$$\hat{\Sigma} = \frac{\mathbf{E}^\top \mathbf{E}}{m - n} \tag{12}$$

The overall standard deviation of the LS fit is obtained as

$$\hat{\sigma} = \sqrt{\frac{\mathrm{tr}(\hat{\Sigma})}{r}} \tag{13}$$

where $\mathrm{tr}(.)$ is the trace operator. For this application $r = 1$, and hence $\hat{\Sigma} = \hat{\sigma}^2$ is the estimated variance of the fit. The estimate $\hat{\sigma}$ serves as a global measure of model fit quality on the training data. The code for the function is provided in Appendix. Figure 7 illustrates the estimated standard deviation of the least-squares fit over different iterations. In this example, the estimate converges to approximately 0.01, which matches the standard deviation of the noise added to the simulated data.

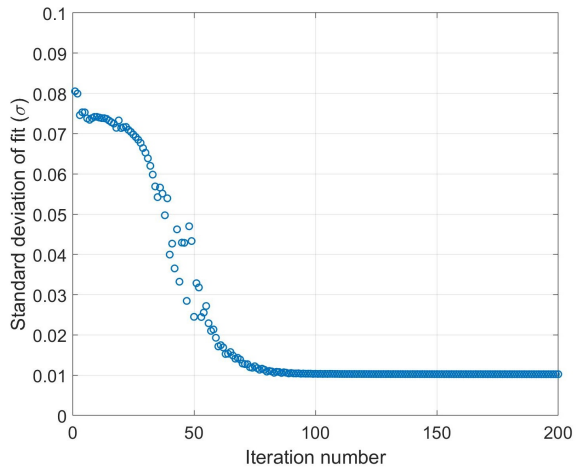


Figure 7. Estimated standard deviation of the least-squares fit.

After optimizing the network, the model of observation equations can be established. In principle, the existing least squares theory can then be directly applied to this model. For example, hypothesis testing could be used to identify outliers in the data. However, this functionality has not been implemented in the current version. Instead, we simply plot the least squares estimate of the residuals (Fig. 8). For this example, the residuals exhibit more or less the same variability, as expected from the simulated data, and are consistent with the estimated standard deviation of $\hat{\sigma} = 0.01$.

```
figure
plot(MAT.E,'o')
hold on
xlabel('Observation ID')
ylabel('Least squares residuals')
box on, grid on
```

Step 3: To evaluate the predictive performance of the trained network, a testing dataset was generated. Since the underlying mathematical function employed to simulate the data is known, the true function value at any given point can simply be determined. In this step, we generate a regular grid of testing points (u, v), calculated the corresponding true function values $f(u, v)$, and then used the trained model to predict these values based on the input features D_t (i.e., the extracted features of 'Case 1' or 'Case 2' using the coordinates of the grid points). This is performed using the `lsbdl_sd_test` function, which is described in Appendix. The predicted outputs were then compared to the true values to assess the model's performance. The

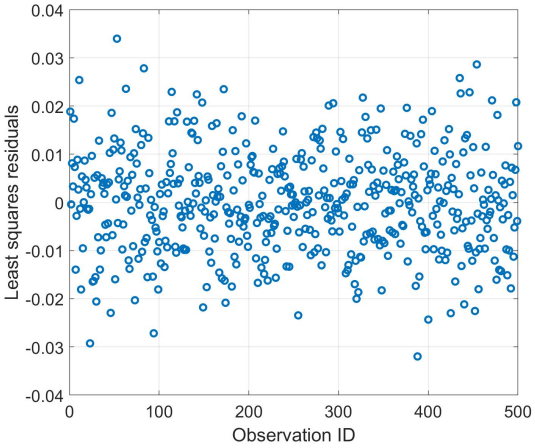


Figure 8. Least squares estimate of residuals.

difference between the predicted and true values provides a direct measure of the prediction error. A surface plot of the predicted values is illustrated in Fig. 9, and the prediction errors in Fig. 10, which are generally small. In a typical run, the root mean square error (RMSE) is 0.0021, which is significantly lower than the noise level added to the simulated data.

```
% prepare testing data
[Ut, Vt] = meshgrid(-2:0.1:2, -2:0.1:2);
Yt = (Ut.^2 + Vt.^2) .* exp(-Ut.^2 - Vt.^2);
[mr nr] = size(Ut);
ut = reshape(Ut,mr*nr,1);
vt = reshape(Vt,mr*nr,1);
yt = reshape(Yt,mr*nr,1);
mt = length(ut);
if Case==1
    Dt = [ones(mt,1) ut vt];
else
    Dt = [ones(mt,1) ut vt ut.*vt ut.^2 vt.^2];
end
% Testing step
MAT.Dt = Dt;
MAT.Yt = yt;
MAT = lsbdl.testing(MAT, P, Activation);
```

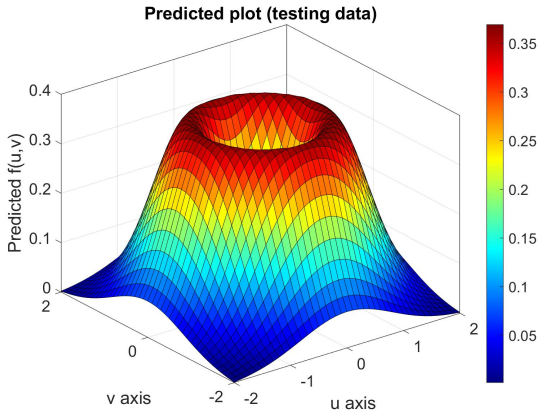


Figure 9. Prediction of the mathematical function in Fig. 5.

4. Conclusions

LSBDL leverages flexibility of deep learning to capture complex patterns to iteratively train a design matrix that models nonlinear relationships between input and output. In this method,

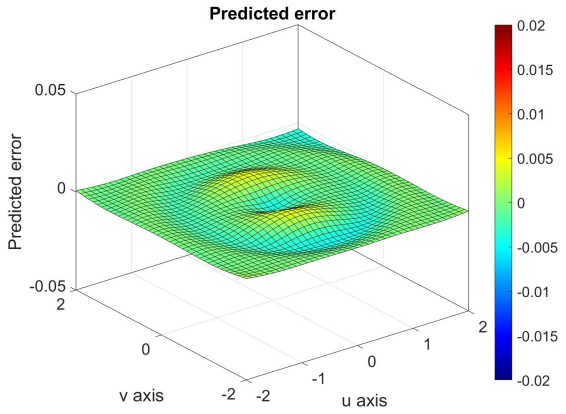


Figure 10. Prediction error of mathematical function in Fig. 5.

the design matrix is built through successive iterations to enhance the match between input and output. From the perspective of explainable AI (XAI), LSBDL enhances the interpretability and transparency of deep learning models. A gradient-based optimization technique, namely steepest descent (SD), has been employed to formulate LSBDL. It offers several attractive features, including the ability to incorporate heterogeneous information by integrating the covariance matrix of observations during training. This contribution focused on implementing a single-layer LSBDL model using the SD method. Its potential applications span geoscience and aviation domains, including GNSS, GRACE, aviation acoustics, and operational challenges that fall beyond the scope of traditional linear model theory.

References

Amiri-Simkooei, A., 2009. Noise in multivariate GPS position time-series. *Journal of Geodesy*, 83(2), 175–187.

Amiri-Simkooei, A., Tiberius, C., Lindenbergh, R., 2024. Deep learning in standard least-squares theory of linear models: Perspective, development and vision. *Engineering Applications of Artificial Intelligence*, 138, 109376.

Arrieta, A. B., Díaz-Rodríguez, N., Del Ser, J., Bennetot, A., Tabik, S., Barbado, A., García, S., Gil-López, S., Molina, D., Benjamins, R. et al., 2020. Explainable Artificial Intelligence (XAI): Concepts, taxonomies, opportunities and challenges toward responsible AI. *Information Fusion*, 58, 82–115.

Bishop, C. M., 1995. *Neural networks for pattern recognition*. Oxford University Press.

Choi, R. Y., Coyner, A. S., Kalpathy-Cramer, J., Chiang, M. F., Campbell, J. P., 2020. Introduction to machine learning, neural networks, and deep learning. *Translational vision science & technology*, 9(2), 14–14.

Costarelli, D., Spigler, R. et al., 2013. Constructive approximation by superposition of sigmoidal functions. *Anal. Theory Appl*, 29(2), 169–196.

Cybenko, G., 1989. Approximation by superpositions of a sigmoidal function. *Mathematics of Control, Signals and Systems*, 2(4), 303–314.

Emmert-Streib, F., Yang, Z., Feng, H., Tripathi, S., Dehmer, M., 2020. An introductory review of deep learning for prediction models with big data. *Frontiers in Artificial Intelligence*, 3, 4.

Haykin, S., 1994. *Neural networks: a comprehensive foundation*. Prentice Hall PTR.

Jordan, M. I., Mitchell, T. M., 2015. Machine learning: Trends, perspectives, and prospects. *Science*, 349(6245), 255–260.

McKinnell, L.-A., 2008. Using neural networks to determine the optimum solar input for the prediction of ionospheric parameters. *Advances in space research*, 42(4), 634–638.

Najafabadi, M. M., Villanustre, F., Khoshgoftaar, T. M., Seliya, N., Wald, R., Muharemagic, E., 2015. Deep learning applications and challenges in big data analytics. *Journal of Big Data*, 2(1), 1–21.

Sabzehee, F., Amiri-Simkooei, A., Iran-Pour, S., Vishwakarma, B. D., Kerachian, R., 2023. Enhancing spatial resolution of GRACE-derived groundwater storage anomalies in Urmia catchment using machine learning downscaling methods. *Journal of Environmental Management*, 330, 117180.

Sabzehee, F., Farzaneh, S., Sharifi, M. A., Akhoondzadeh, M., 2018. TEC Regional Modeling and prediction using ANN method and single frequency receiver over IRAN. *Annals of Geophysics*, 61(1), GM103–GM103.

Teunissen, P. J. G., 1990. Nonlinear least-squares. *Manuscripta Geodetica*, 15(3), 137–150.

Teunissen, P. J. G., 2000. *Testing theory: an introduction*. Delft University Press, Website: http://www.vssd.nl. Series on Mathematical Geodesy and Positioning.

Teunissen, P. J. G., 2018. Distributional theory for the DIA method. *Journal of Geodesy*, 92(1), 59–80.

Tikhonov, A. N., 1963. Solution of incorrectly formulated problems and the regularization method. *Soviet Math.*, 4, 1035–1038.

Williams, P., 2017. Assessing collaborative learning: big data, analytics and university futures. *Assessment & Evaluation in Higher Education*, 42(6), 978–989.

APPENDIX

The Matlab code of the four functions of LSBDL is presented.

Activation function:

```
function [A, Ap] = activation_function(I, actp,
    Activation)
if nargin < 3
    % Default activation function
    Activation = 'sigmoid';
end
if nargin < 2
    % Default activation parameter
    actp = 1;
end
% Scale the input
I = actp * I;
choice = lower(Activation);
switch choice
    case {'sigmoid'}
        A = 1./(1+exp(-I));
        Ap = actp * (A-A.^2);
    case {'softmax'}
        A = softmax(I')'; % Approximation
        Ap = actp * (A-A.^2);
    case {'relu'}
        A = max(0,I);
        Ap = actp * sign(A);
    case {'softsign'}
        A = I./(1+abs(I));
        Ap = actp * 1./(1+abs(I)).^2;
    case {'tanh'}
        A = (exp(I)-exp(-I))./(exp(I)+exp(-I));
        Ap = actp * (1-A.^2);
    case {'cubic'}
        A = I.^3;
        Ap = actp * 3*I.^2;
    case {'softplus'}
        A = log(1+exp(I));
        Ap = actp * 1./(1+exp(-I));
    otherwise
        error('Activation function is not
            recognized!');
end
end
```

Initialization function:

```
function MAT = lsbdl_sd_initialize(MAT,P,
    Activation)
% number of features
k = P.k;
% number of neurons (unknowns in X)
n = P.n;
% regularization parameters
kappa = P.kappa;
% activation parameter
actp = P.actp;
% feature matrix
D = MAT.D;
% initialize W
W = 2*rand(k,n)-1;
W = 0.1*W/mean(std(MAT.D*W));
MAT.W = W;
% compute design matrix and its derivative
[A, Ap] = activation_function(D*W, actp,
    Activation);
MAT.A = A; MAT.Ap = Ap;
% least-squares estimate of X
MAT.X = inv(A'*A + kappa*eye(n))*A'*MAT.Y;
% least-squares estimate of Y
MAT.Yh = A*MAT.X;
% least-squares estimate of E
MAT.E = MAT.Y - MAT.Yh;
% initialize dW
MAT.dW = zeros(k,n);
end
```

Optimization function:

```
function MAT = lsbdl_sd_optimize(MAT, P,
    Activation)
% number of observations (data points in Y)
m = P.m;
% number of neurons (unknowns in X)
k = P.k;
% number of features
n = P.n;
% number of output variables
r = P.r;
% softening parameter
spar = P.spar;
% regularization parameters
kappa = P.kappa;
% learning rate parameter
alpha = P.alpha;
% momentum parameter
mu = P.mu;
% activation parameter
actp = P.actp;
% iteration number
it = P.it;
% compute the gradient
G = -MAT.D'*((MAT.E*MAT.X').*MAT.Ap);
% soften the gradient
G = ((abs(G)).^spar) .* sign(G);
% compute weight corrections (considering
    momentum)
MAT.dW = -alpha*G + mu*MAT.dW;
% updates weight W
MAT.W = MAT.W + MAT.dW;
% obtain the design matrix and its derivative
[MAT.A, MAT.Ap] = activation_function(MAT.D*MAT.W
    , actp, Activation);
% least-squares estimate of unknowns X
MAT.X = inv(MAT.A'*MAT.A + kappa*eye(n))*MAT.A'*
    MAT.Y;
% least-squares estimate of observations Y
MAT.Yh = MAT.A*MAT.X;
% least-squares estimate of residuals E
MAT.E = MAT.Y - MAT.Yh;
E = MAT.E;
% estimated standard deviation of LS fit
MAT.Std(it) = sqrt(trace(E'*E/(m-n))/r);
end
```

Testing function:

```
function MAT = lsbdl_sd_test(MAT,P,Activation)
[MAT.At, MAT.Apt] = activation_function(MAT.Dt*
    MAT.W, P.actp, Activation);
% predicted test data
MAT.Yht = MAT.At*MAT.X;
% prediction error
MAT.Et = MAT.Yt-MAT.Yht;
mt = size(MAT.Et,1);
% RMSE of predicted results
MAT.RMSE = sqrt(trace(MAT.Et'*MAT.Et/mt)/P.r);
end
```

This contribution has been peer-reviewed. The double-blind peer-review was conducted on the basis of the full paper.
https://doi.org/10.5194/isprs-annals-X-4-W8-2025-85-2026 | © Author(s) 2026. CC BY 4.0 License.