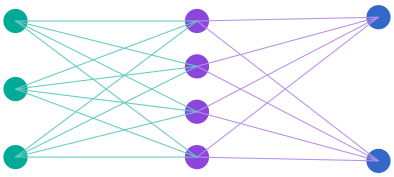


Artificial Intelligence Applications in Matter Science

Workshop Educational Document

Regression using an artificial neuron: Concepts and applications

Date: December 09, 2025

Instructor: Dr. Samir Kenouche

Institution: LCA Laboratory

Team: Multi-scale quantum modeling of chemical systems

Department of matter science – University of Biskra

The Perceptron as a Linear Regressor: Simple and High-Level Matrix Formulation - Application in Materials Physics and Chemistry.

Dr. Samir Kenouche

Workshop on December 9, 2025

The advent of Machine Learning has revolutionized many scientific fields, particularly materials chemistry and physics. The study and discovery of new materials with optimized properties are traditionally long and costly processes, relying heavily on experimentation and intensive *ab initio* simulations. Neural Network (NN) Regression is a powerful sub-domain of machine learning that plays a central role in accelerating this process. An Artificial Neural Network is a computational model inspired by the structure of the brain, composed of layers of interconnected neurons. Regression involves using this model to learn the complex functional relationship between a set of input variables (or descriptors) describing a material's composition, structure, or synthesis conditions, and one or more continuous output variables (or properties) such as formation energy, band gap, density, hardness, or transition temperature. The importance of this approach is twofold:

1. **Rapid Prediction and High-Throughput Screening:** Trained NN models can accurately predict the properties of thousands of potential materials in a fraction of a second, without requiring long *DFT* (Density Functional Theory) calculations. This enables rapid virtual screening, drastically reducing the search space for promising materials.
2. **Discovery of Complex Relationships:** NNs are universal function approximators. They are capable of capturing nonlinear relationships and complex interactions between material descriptors that traditional linear models or even simple analytical functions could not effectively model. They thus help to identify trends and design rules for materials.

Internally, the training and prediction process of a neural network involves a series of multiplications and additions, particularly between the inputs ($\mathbf{X}$) and the network's weights ($\mathbf{W}$). The use of the matrix formulation provides crucial advantages for implementing neural networks:

1. **Computational Efficiency (Vectorization):** Matrix operations (like $\mathbf{XW}$) are highly optimized. Modern libraries (such as NumPy, TensorFlow, PyTorch) exploit vectorization and parallel computing on GPUs (Graphics Processing Units) to perform thousands, or even millions, of operations simultaneously. Simple, sequentially executed summation would be far too slow for large datasets and deep networks.

2. **Clarity and Compactness:** Matrix notation (e.g., $\mathbf{Z} = \mathbf{XW} + \mathbf{B}$) concisely expresses all the operations of an entire neuron layer. It simplifies the mathematical expression of the backpropagation algorithm necessary for training, which relies on calculating the gradients of the weight matrices.

3. **Standardization:** Matrix linear algebra is the universal language for implementing deep learning algorithms, facilitating the development and exchange of models among researchers.

It is worth noting that the various theoretical concepts included in this document are mainly drawn from the references cited here.

References

- Butler K. T., Oviedo F., Canepa P. *Machine Learning in Materials Science*. American Chemical Society. DOI:10.1021/acsinfocus.7e5033.

- Mueller T., Gilad A., Ramprasad K. *Machine Learning in Materials Science: Recent Progress and Emerging Applications*. Reviews in Computational Chemistry. https://doi.org/10.1002/9781119148739.ch4.

- Jain A. *Machine learning in materials research: Developments over the last decade and challenges for the future*. Current Opinion in Solid State and Materials Science. https://doi.org/10.1016/j.cossms.2024.101189.

Copyright Notice

Sub-workshop 1: Regression Task with Simple Formulation

The perceptron is commonly used for classification, but it can also perform linear regression when the activation function is **linear** (identity function) and the objective is to minimize the Mean Squared Error (MSE). This report provides the mathematical formulation, a detailed numerical example, and Python code to demonstrate training a perceptron as a linear regressor.

1 Learning Steps: Mathematical formulation

This section is devoted to outlining the key steps for implementing a perceptron in the case of simple regression. Participants are not required to literally learn equations, but instead to apprehend them and how to use them.

Step 1 Forward Model (Prediction)

The perceptron regressor produces a continuous prediction by applying a **linear transformation** to the input vector $x \in \mathbb{R}^d$.

$$\hat{y} = f(x) = w^\top x + b,$$

where:

- $w \in \mathbb{R}^d$ is the weight vector,
- $b \in \mathbb{R}$ is the bias,
- $\hat{y}$ is the predicted real-valued output.

This step simply computes a weighted sum of input features plus a bias.

Step 2 Loss Function (Error Measurement): Training

To train the model, we quantify how far predictions are from true values. For regression, we use the **Mean Squared Error (MSE)**:

$$\mathcal{L}(w, b) = \frac{1}{2N} \sum_{i=1}^{N} (\hat{y}_i - y_i)^2 = \frac{1}{2N} \sum_{i=1}^{N} (w^\top x_i + b - y_i)^2.$$

The loss is small when predictions are close to targets, large otherwise. Minimizing this loss drives the learning process.

Interested participants are invited to ask questions about any theoretical aspect relating to pure mathematical concepts or about the statistical formulas presented in this section. The gradient descent optimization procedure is a key aspect of neural network learning. This procedure will be studied in detail using simple examples in the upcoming practical activities.

Step 3 Gradient Computation: Parameters Optimisation

To minimize the loss, we compute its derivatives with respect to the parameters. Let the prediction error be:

$$e_i = \hat{y}_i - y_i = w^\top x_i + b - y_i.$$

Gradient with respect to weights:

$$\nabla_w \mathcal{L} = \frac{1}{N} \sum_{i=1}^{N} (\hat{y}^{(i)} - y^{(i)})\, x_j^{(i)}$$

Gradient with respect to bias:

$$\nabla_b \mathcal{L} = \frac{1}{N} \sum_{i=1}^{m} (\hat{y}^{(i)} - y^{(i)})$$

These gradients indicate how each parameter contributes to the error and in which direction it must be adjusted to reduce the loss.

Step 4 Parameter Update (Gradient Descent)

Using a learning rate $\eta > 0$, we update the parameters by moving them **opposite to the gradient**:

$$w^{(n+1)} = w^{(n)} - \eta\, \nabla_w \mathcal{L}, \qquad b^{(n+1)} = b^{(n)} - \eta\, \nabla_b \mathcal{L}.$$

Substituting the gradients:

$$w^{(n+1)} = w^{(n)} - \eta \cdot \left[\frac{1}{N} \sum_{i=1}^{N} (\hat{y}^{(i)} - y^{(i)}) x_j^{(i)} \right]$$

$$b^{(n+1)} = b^{(n)} - \eta \cdot \left[\frac{1}{N} \sum_{i=1}^{m} (\hat{y}^{(i)} - y^{(i)}) \right]$$

This iterative update progressively reduces the loss and trains the perceptron regressor.

At first glance, these equations may seem complicated to non-specialists. Nevertheless, their application is relatively simple, as we will demonstrate in the next practical activity. Participants are encouraged to ask questions at any time during the activity.

Activity 1: Simple Regression

Numerical Example + Python

In this first activity, we will apply the perceptron learning strategy for simple regression step by step. This task is important for understanding how a machine learning algorithm works and grasping the role of hyperparameters. This will greatly facilitate generalization to neural networks with multiple layers.

2 Diagram of the Linear Perceptron

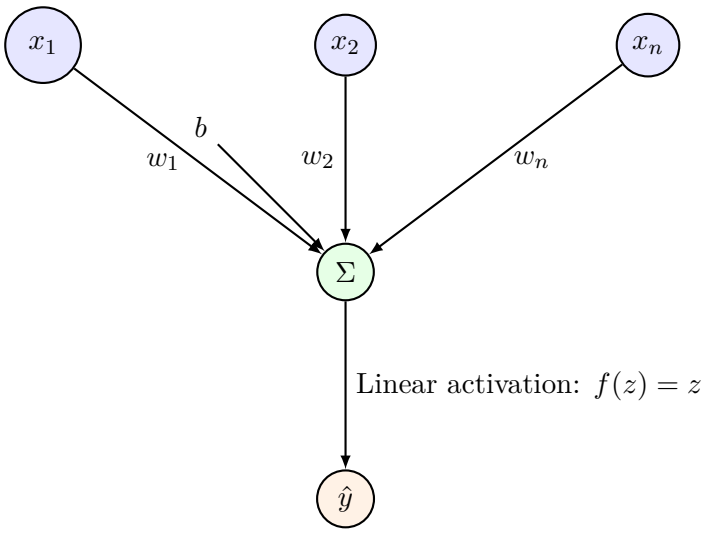


Figure 1: Linear Perceptron Architecture for Regression

3 Algorithm Summary

1. Initialize $\mathbf{w}$ and b (zeros or small random values).
2. For each iteration:
   - Compute predicted output $\hat{y}^{(i)} = \mathbf{w}^T \mathbf{x}^{(i)} + b$.
   - Compute the gradient of the loss function.
   - Update the weights and bias using gradient descent.
3. Stop when the loss converges or the maximum number of epochs is reached.

This perceptron will be studied step by step using a simple example. The calculations will be done step by step, and feel free to ask questions about any aspect.

4 Detailed Numerical Example

We consider the following training data, which follows a perfect linear relation $y = 2x$:

i	x_i	y_i
1	1	2
2	2	4
3	3	6

We use:

$$\eta = 0.01, \quad w^{(0)} = 0, \quad b^{(0)} = 0$$

The model is:

$$\hat{y}_i = wx_i + b$$

Iteration 1

$$\hat{y} = [0, 0, 0], \quad e = [-2, -4, -6]$$

$$\frac{\partial \mathcal{L}}{\partial w} = \frac{1}{3}(-2 \times 1 - 4 \times 2 - 6 \times 3) = -9.333, \quad \frac{\partial \mathcal{L}}{\partial b} = \frac{-2-4-6}{3} = -4$$

$$w^{(1)} = 0 - 0.01(-9.333) = 0.0933, \quad b^{(1)} = 0 - 0.01(-4) = 0.04$$

Iteration 2

$$w = 0.0933, \quad b = 0.04$$

$$\hat{y} = [0.133, 0.227, 0.320], \quad e = [-1.867, -3.773, -5.680]$$

$$\frac{\partial \mathcal{L}}{\partial w} = -8.818, \quad \frac{\partial \mathcal{L}}{\partial b} = -3.773$$

$$w^{(2)} = 0.1815, \quad b^{(2)} = 0.0777$$

Iteration 3

$$w = 0.1815, \quad b = 0.0777$$

$$\hat{y} = [0.259, 0.440, 0.621], \quad e = [-1.741, -3.560, -5.379]$$

$$\frac{\partial \mathcal{L}}{\partial w} = -8.333, \quad \frac{\partial \mathcal{L}}{\partial b} = -3.560$$

$$w^{(3)} = 0.2648, \quad b^{(3)} = 0.1133$$

After about 1000 epochs:

$$w^{(1000)} \approx 1.9995, \quad b^{(1000)} \approx 0.001$$

and the predictions become nearly exact.

x	y_{true}	$\hat{y}$
1	2	2.00
2	4	4.00
3	6	6.00

5 Python Implementation

Complete Python Implementation and Plotting

```
import numpy as np
import matplotlib.pyplot as plt
# Samir KENOUCHE - 28/11/2025

# Training data
X = np.array([[1], [2], [3]])
y = np.array([[2], [4], [6]])

# Initialization
w, b = 0.0, 0.0
eta = 0.01
epochs = 1000
m = len(X)
losses = []

# Training loop
for epoch in range(epochs):
    y_pred = X * w + b
    error = y_pred - y
    loss = (1/(2*m)) * np.sum(error**2)
    losses.append(loss)

    dw = (1/m) * np.sum(error * X)
    db = (1/m) * np.sum(error)

    w = w - eta * dw
    b = b - eta * db

print(f"Final w = {w:.4f}, b = {b:.4f}")

# Plot fitted line
plt.figure(figsize=(6,4))
plt.scatter(X, y, color='blue', label='Training data')
plt.plot(X, X*w + b, color='red', label='Fitted line')
plt.xlabel("x"); plt.ylabel("y")
plt.legend(); plt.grid(True)
plt.title("Perceptron as Linear Regressor")
plt.show()

# Plot loss curve
plt.figure(figsize=(6,4))
plt.plot(losses)
plt.title("Convergence of Loss (MSE)")
plt.xlabel("Epoch"); plt.ylabel("Loss")
plt.grid(True)
plt.show()
```

6 Conclusion

In this activity, we demonstrated that the perceptron model can be used for linear regression by adopting a linear activation function and minimizing the mean squared error. Gradient descent successfully recovers the parameters of a linear model ($w \approx 2$, $b \approx 0$) when trained on simple data following $y = 2x$.

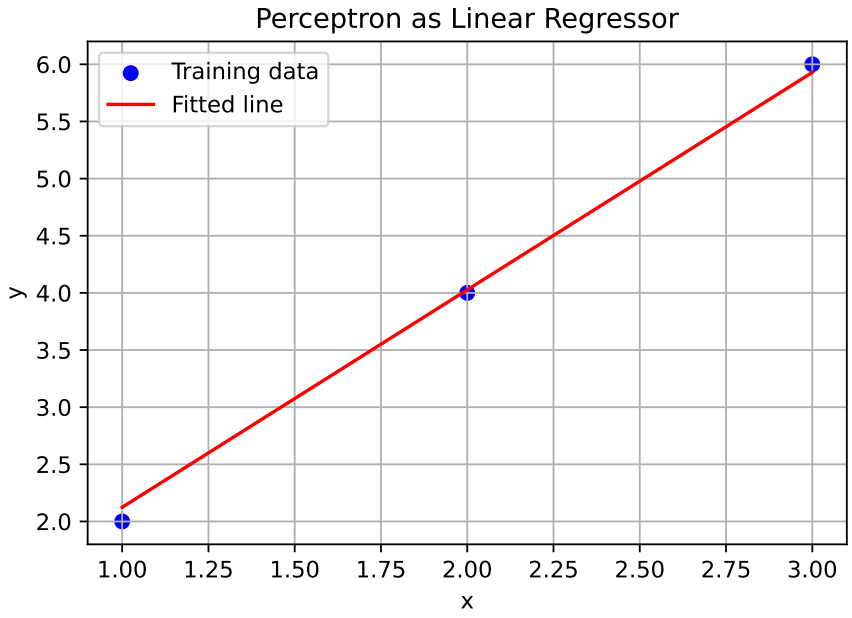


Figure 2: Perceptron as Linear Regressor

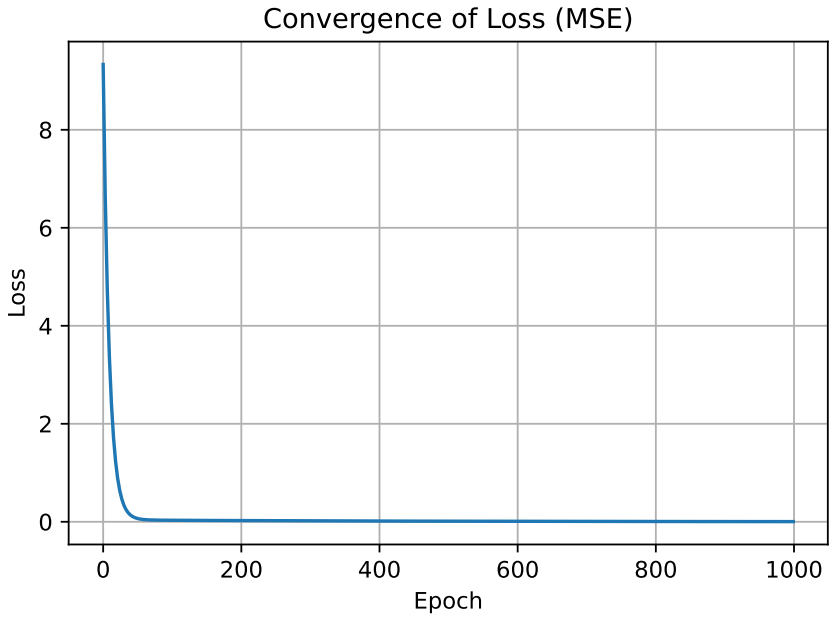


Figure 3: Convergence of Loss (MSE)

The appendix below is not part of the practical activity and is provided as a theoretical supplement for participants interested in understanding parameter adjustment using the least squares method. Participants who already have this knowledge or are **not interested in such detail may skip this section and move directly to the next practical activity**.

Appendix. Additional information

A Mathematical Derivation of the Least Squares Parameters for a Linear Model

A.1 Model and Objective Function

Consider a set of n observed data points (x_i, y_i), $i = 1, 2, \ldots, n$. The simple linear regression model is written as

$$y_i = \beta_0 + \beta_1 x_i + \varepsilon_i, \tag{1}$$

where β_0 and β_1 are unknown parameters (intercept and slope, respectively), and ε_i represents the random error associated with each observation.

The least squares principle seeks to determine β_0 and β_1 that minimize the *sum of squared residuals*:

$$S(\beta_0, \beta_1) = \sum_{i=1}^{n} (y_i - \beta_0 - \beta_1 x_i)^2 . \tag{2}$$

A.2 First-Order Conditions

To find the minimum of S, we differentiate with respect to β_0 and β_1 and set the derivatives to zero.

Derivative with respect to β_0:

$$\begin{aligned} \frac{\partial S}{\partial \beta_0} &= \sum_{i=1}^{n} 2 (y_i - \beta_0 - \beta_1 x_i)(-1) \\ &= -2 \sum_{i=1}^{n} (y_i - \beta_0 - \beta_1 x_i) . \end{aligned} \tag{3}$$

Setting this derivative to zero yields

$$\sum_{i=1}^{n} y_i - n\beta_0 - \beta_1 \sum_{i=1}^{n} x_i = 0. \tag{4}$$

Derivative with respect to β_1:

$$\begin{aligned} \frac{\partial S}{\partial \beta_1} &= \sum_{i=1}^{n} 2 (y_i - \beta_0 - \beta_1 x_i)(-x_i) \\ &= -2 \sum_{i=1}^{n} x_i (y_i - \beta_0 - \beta_1 x_i) . \end{aligned} \tag{5}$$

Setting this derivative to zero gives

$$\sum_{i=1}^{n} x_i y_i - \beta_0 \sum_{i=1}^{n} x_i - \beta_1 \sum_{i=1}^{n} x_i^2 = 0. \tag{6}$$

Equations (4) and (6) are known as the **normal equations** of simple linear regression:

$$\boxed{\begin{aligned} n\beta_0 + \beta_1 \sum x_i &= \sum y_i, \\ \beta_0 \sum x_i + \beta_1 \sum x_i^2 &= \sum x_i y_i. \end{aligned}} \tag{7}$$

A.3 Solution for the Parameters

From the first equation of (7),

$$\beta_0 = \frac{1}{n}\sum_{i=1}^{n} y_i - \frac{\beta_1}{n}\sum_{i=1}^{n} x_i. \tag{8}$$

Introduce the sample means

$$\bar{x} = \frac{1}{n}\sum_{i=1}^{n} x_i, \qquad \bar{y} = \frac{1}{n}\sum_{i=1}^{n} y_i, \tag{9}$$

so that

$$\boxed{\beta_0 = \bar{y} - \beta_1 \bar{x}.} \tag{10}$$

Substitute (10) into the second normal equation of (7):

$$(\bar{y} - \beta_1\bar{x})\sum x_i + \beta_1 \sum x_i^2 = \sum x_i y_i,$$

$$n\bar{x}\bar{y} - n\beta_1\bar{x}^2 + \beta_1 \sum x_i^2 = \sum x_i y_i. \tag{11}$$

Rearranging for β_1,

$$\beta_1 \left(\sum x_i^2 - n\bar{x}^2\right) = \sum x_i y_i - n\bar{x}\bar{y}, \tag{12}$$

and thus

$$\boxed{\hat{\beta}_1 = \frac{\displaystyle\sum_{i=1}^{n} x_i y_i - n\bar{x}\bar{y}}{\displaystyle\sum_{i=1}^{n} x_i^2 - n\bar{x}^2}.} \tag{13}$$

A.4 Centered Variable Form

Note that

$$\sum_{i=1}^{n}(x_i - \bar{x})(y_i - \bar{y}) = \sum x_i y_i - n\bar{x}\bar{y}, \quad \sum_{i=1}^{n}(x_i - \bar{x})^2 = \sum x_i^2 - n\bar{x}^2.$$

Hence Equation (13) can be rewritten compactly as

$$\boxed{\hat{\beta}_1 = \frac{\displaystyle\sum_{i=1}^{n}(x_i - \bar{x})(y_i - \bar{y})}{\displaystyle\sum_{i=1}^{n}(x_i - \bar{x})^2},} \tag{14}$$

and, substituting back into (10),

$$\boxed{\hat{\beta}_0 = \bar{y} - \hat{\beta}_1 \bar{x}.} \tag{15}$$

A.5 Verification of Minimization

The Hessian matrix of $S(\beta_0, \beta_1)$ is

$$H = 2\begin{pmatrix} n & \sum x_i \\ \sum x_i & \sum x_i^2 \end{pmatrix}. \tag{16}$$

Since H is positive definite whenever $\sum(x_i - \bar{x})^2 > 0$, the stationary point found above corresponds to a *minimum* of the sum of squared errors.

A.6 Final Least Squares Estimates

The least squares estimates of the parameters in the linear model (1) are therefore:

$$\boxed{\begin{aligned}\hat{\beta}_1 &= \frac{\sum_{i=1}^{n}(x_i-\bar{x})(y_i-\bar{y})}{\sum_{i=1}^{n}(x_i-\bar{x})^2},\\ \hat{\beta}_0 &= \bar{y}-\hat{\beta}_1\bar{x}.\end{aligned}} \tag{17}$$

The fitted regression line is finally expressed as

$$\hat{y}_i = \hat{\beta}_0 + \hat{\beta}_1 x_i, \tag{18}$$

and the residuals are

$$\hat{\varepsilon}_i = y_i - \hat{y}_i. \tag{19}$$

A.7 Interpretation

The parameter $\hat{\beta}_1$ represents the estimated change in y for a one-unit change in x, while $\hat{\beta}_0$ represents the fitted value of y when $x = 0$. Notably, the regression line always passes through the point $(\bar{x}, \bar{y})$.

Summary of the Least Squares Estimation Results

Model: $y_i = \beta_0 + \beta_1 x_i + \varepsilon_i$

The least squares method minimizes

$$S(\beta_0, \beta_1) = \sum_{i=1}^{n}(y_i - \beta_0 - \beta_1 x_i)^2.$$

The resulting estimators are:

$$\hat{\beta}_1 = \frac{\sum_{i=1}^{n}(x_i-\bar{x})(y_i-\bar{y})}{\sum_{i=1}^{n}(x_i-\bar{x})^2}, \tag{20}$$

$$\hat{\beta}_0 = \bar{y} - \hat{\beta}_1\bar{x}. \tag{21}$$

Hence, the fitted regression line is:

$$\boxed{\hat{y}_i = \hat{\beta}_0 + \hat{\beta}_1 x_i}$$

and the residuals are:

$$\hat{\varepsilon}_i = y_i - \hat{y}_i.$$

Properties of the Least Squares Estimates

- The regression line passes through $(\bar{x}, \bar{y})$.
- Residuals are orthogonal to the fitted line:

$$\sum_i \hat{\varepsilon}_i = 0, \qquad \sum_i x_i\hat{\varepsilon}_i = 0.$$

- The slope $\hat{\beta}_1$ measures the average change in y for one unit change in x.

Activity 2: Gradient descent by example

Gradient Descent in 1D

Gradient Descent is the fundamental algorithm that enables learning in neural networks. Its paramount importance lies in its ability to minimize the cost function (or loss function), which quantifies the error between the network's predictions and the actual target values. The algorithm works by calculating the gradient of this cost function with respect to the network's weights and biases using a technique called backpropagation. This gradient indicates the direction of the steepest ascent; consequently, Gradient Descent iteratively adjusts the model's parameters in the exact opposite direction (down the "slope"). This process ensures that, with each iteration, the network becomes progressively more accurate, allowing the models to effectively learn and map complex, non-linear relationships within the data, which is crucial for tasks like regression and classification. This section is dedicated to this purpose with a minimum of math.

B Function and Gradient

We consider the function:

$$f(x) = x^2 - 4x + 5$$

Derivative (gradient):

$$f'(x) = 2x - 4$$

Gradient descent update:

$$x_{k+1} = x_k - \eta f'(x_k)$$

We choose:

$$x_0 = 0, \quad \eta = 0.1$$

Gradient Descent Formula

The update rule of gradient descent is:

$$x_{k+1} = x_k - \eta f'(x_k)$$

- x_k: Current point
- $f'(x_k)$: Derivative (slope) at x_k, indicates the direction of steepest ascent
- η: Learning rate (step size), controls how far we move in the opposite direction of the gradient
- x_{k+1}: Next point after one gradient step

Intuition

Gradient descent moves **opposite to the slope** to find a minimum. For convex functions, it guarantees convergence to the global minimum if η is chosen appropriately. The stopping criterion is:

$$|f'(x_k)| < \text{tolerance}$$

C Detailed Calculations for First Three Iterations

Step 1 to 3

Iteration 0 → 1

$$x_0 = 0, \quad f'(0) = -4, \quad x_1 = 0 - 0.1(-4) = 0.4, \quad f(x_1) = 3.56$$

Iteration 1 → 2

$$x_1 = 0.4, \quad f'(0.4) = -3.2, \quad x_2 = 0.4 - 0.1(-3.2) = 0.72, \quad f(x_2) = 2.6384$$

Iteration 2 → 3

$$x_2 = 0.72, \quad f'(0.72) = -2.56, \quad x_3 = 0.72 - 0.1(-2.56) = 0.976, \quad f(x_3) = 2.048$$

D Iteration Table

The table below summarizes the numerical values of a few iterations obtained with one-dimensional gradient descent.

Iterations from Step 1 to 19

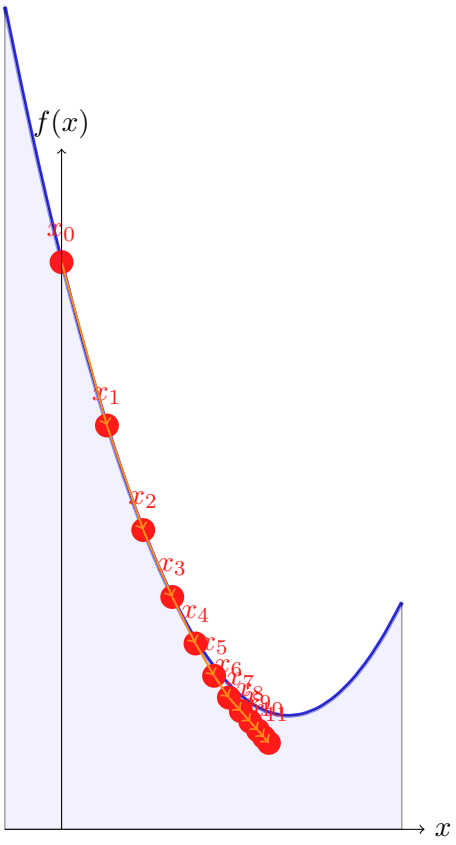
Iteration k	x_k	$f(x_k)$	$f'(x_k)$
0	0	5	-4
1	0.4	3.56	-3.2
2	0.72	2.6384	-2.56
3	0.976	2.048	-2.048
4	1.1808	1.638	-1.638
5	1.34464	1.3535	-1.3107
6	1.4753	1.161	-1.0494
7	1.5802	1.043	-0.8396
8	1.6642	0.948	-0.6716
9	1.7313	0.872	-0.5374
10	1.7849	0.811	-0.4301
11	1.8279	0.762	-0.3442
12	1.8623	0.723	-0.2754
13	1.8898	0.692	-0.2204
14	1.9118	0.667	-0.1764
15	1.9278	0.648	-0.1444
16	1.9422	0.632	-0.1156
17	1.9538	0.619	-0.0924
18	1.9629	0.608	-0.0742
19	1.9704	0.600	-0.0592

E Python Code

1D Gradient Descent Python Code

```
# Dr. Samir Kenouche -- 28/11/2025
def f(x):
    return x**2 - 4*x + 5

def grad_f(x):
    return 2*x - 4

x = 0.0
eta = 0.1
tolerance = 1e-4
iteration = 0

print("Iteration | x | f(x) | grad_f(x)")

while abs(grad_f(x)) > tolerance:
    print("Iteration " + str(iteration) +
          " : x=" + str(round(x,6)) +
          ", f=" + str(round(f(x),6)) +
          ", grad=" + str(round(grad_f(x),6)))
    x = x - eta * grad_f(x)
    iteration += 1

print("Minimum approximated at x = " + str(x) + ", f(x) = " + str(f(x)))
```

F Gradient Descent Plot



G Python Code with Plot

1D Gradient Descent Python Code and plotting

```
# Dr. Samir Kenouche -- 27/11/2025

import matplotlib.pyplot as plt
import numpy as np

# Define function and derivative
def f(x):
    return x**2 - 4*x + 5
def grad_f(x):
    return 2*x - 4

# Gradient Descent parameters
x = 0.0
eta = 0.1
tolerance = 1e-4
max_iterations = 20
points = [x]

# Gradient Descent loop
print("Iteration | x | f(x) | grad_f(x)")
for iter in range(max_iterations):
    g = grad_f(x)
    if abs(g) < tolerance:
        break
    x = x - eta * g
    points.append(x)

# Plot the function and iterations
x_vals = np.linspace(-0.5, 3, 400)
y_vals = f(x_vals)

plt.figure(figsize=(8,5))
plt.plot(x_vals, y_vals, color='blue', lw=2,
         label='f(x) = x^2 - 4x + 5')
plt.fill_between(x_vals, y_vals, 0, color='blue', alpha=0.1)

# Plot iteration points and labels
for i, xi in enumerate(points):
    plt.scatter(xi, f(xi), color='red')
    plt.text(xi, f(xi)+0.1, f"x_{i}", color='red', ha='center')

# Draw arrows between iterations
for i in range(len(points)-1):
    plt.annotate("", xy=(points[i+1], f(points[i+1])),
                 xytext=(points[i], f(points[i])),
                 arrowprops=dict(arrowstyle="->", color='orange',
                                 lw=1.5))

plt.xlabel("x")
plt.ylabel("f(x)")
plt.title("Gradient Descent on f(x) = x^2 - 4x + 5")
plt.grid(True)
plt.legend()
plt.show()
```

Activity 3: Gradient descent by example

Gradient Descent in 2D

This activity follows the previous one. Indeed, the primary advantage of demonstrating Gradient Descent in 2D rather than 1D is that the 2D visualization truly illustrates the multi-variable nature and optimization challenge faced by neural networks. While the 1D case only shows a simple line minimizing error along a single parameter, the 2D example introduces the concept of a cost surface. This surface clearly shows the existence of a minimum point (the optimal set of parameters) and how the gradient vector, consisting of partial derivatives, points towards the steepest slope. This allows students to visually grasp that the descent must occur simultaneously across multiple weights ($\mathbf{W}$) in the network, making it a far more intuitive and representative model for the dimensionality of real world deep learning problems, where the number of parameters often ranges into the millions.

Function and Gradient

We consider the function:

$$f(x, y) = x^2 + 2y^2 + xy$$

Gradient:

$$\nabla f(x, y) = \begin{bmatrix} 2x + y \\ x + 4y \end{bmatrix}, \quad \|\nabla f(x, y)\| = \sqrt{(2x + y)^2 + (x + 4y)^2}$$

Gradient descent update:

$$\begin{bmatrix} x_{k+1} \\ y_{k+1} \end{bmatrix} = \begin{bmatrix} x_k \\ y_k \end{bmatrix} - \eta \nabla f(x_k, y_k)$$

Stopping criterion:

$$\|\nabla f(x, y)\| \leq \text{tolerance}$$

We use:

$$(x_0, y_0) = (1.0,\ 1.0), \qquad \eta = 0.1\,.$$

H Detailed Calculations for the First Three Iterations

Below are the detailed numerical calculations for the first three iterations. Participants are invited to ask questions about any aspect of the calculation.

Iteration $0 \to 1$

$$\nabla f(1.0, 1.0) = \begin{pmatrix} 2(1)+1 \\ 1+2(1) \end{pmatrix} = \begin{pmatrix} 3 \\ 3 \end{pmatrix}.$$

Update:

$$\begin{pmatrix} x_1 \\ y_1 \end{pmatrix} = \begin{pmatrix} 1 \\ 1 \end{pmatrix} - 0.1 \begin{pmatrix} 3 \\ 3 \end{pmatrix} = \begin{pmatrix} 0.7 \\ 0.7 \end{pmatrix}.$$

$$\|\nabla f(1,1)\| = \sqrt{3^2+3^2} = \sqrt{18}.$$

Iteration $1 \to 2$

Gradient:

$$\nabla f(0.7, 0.7) = \begin{pmatrix} 2(0.7)+0.7 \\ 0.7+2(0.7) \end{pmatrix} = \begin{pmatrix} 2.1 \\ 2.1 \end{pmatrix}.$$

Update:

$$\begin{pmatrix} x_2 \\ y_2 \end{pmatrix} = \begin{pmatrix} 0.7 \\ 0.7 \end{pmatrix} - 0.1 \begin{pmatrix} 2.1 \\ 2.1 \end{pmatrix} = \begin{pmatrix} 0.49 \\ 0.49 \end{pmatrix}.$$

Norm:

$$\|\nabla f(0.7, 0.7)\| = \sqrt{2.1^2+2.1^2} = \sqrt{8.82}.$$

Iteration $2 \to 3$

Gradient:

$$\nabla f(0.49, 0.49) = \begin{pmatrix} 2(0.49)+0.49 \\ 0.49+2(0.49) \end{pmatrix} = \begin{pmatrix} 1.47 \\ 1.47 \end{pmatrix}.$$

Update:

$$\begin{pmatrix} x_3 \\ y_3 \end{pmatrix} = \begin{pmatrix} 0.49 \\ 0.49 \end{pmatrix} - 0.1 \begin{pmatrix} 1.47 \\ 1.47 \end{pmatrix} = \begin{pmatrix} 0.343 \\ 0.343 \end{pmatrix}.$$

Norm:

$$\|\nabla f(0.49, 0.49)\| = \sqrt{1.47^2+1.47^2}.$$

The numerical calculations for these first iterations are summarized in the following table.

k	x_k	y_k	$\nabla f(x_k, y_k)$	$\|\nabla f\|$	$f(x_k, y_k)$
0	1.000	1.000	(3.000, 3.000)	4.2426	3.000
1	0.700	0.700	(2.100, 2.100)	2.9698	1.470
2	0.490	0.490	(1.470, 1.470)	2.0780	0.720
3	0.343	0.343	(1.029, 1.029)	1.4556	0.353

I Python Code

2D Python Gradient Descent

```
# Dr. Samir Kenouche -- 27/11/2025
import numpy as np

def f(x, y):
    return x*x + x*y + y*y

def grad_f(x, y):
    gx = 2*x + y
    gy = x + 2*y
    return gx, gy

eta = 0.1
tolerance = 1e-6
x, y = 1.0, 1.0

gx, gy = grad_f(x, y)
iteration = 0

print("Iteration | x | y | f(x,y)")

while np.linalg.norm([gx, gy]) > tolerance:
    print("Iteration " + str(iteration) +
          " : x=" + str(round(x, 6)) +
          ", y=" + str(round(y, 6)) +
          ", f=" + str(round(f(x, y), 6)))

    x = x - eta * gx
    y = y - eta * gy

    gx, gy = grad_f(x, y)
    iteration += 1

print("Minimum approximated at (x, y) = (" +
      str(x) + ", " + str(y) + ")")
```

J Function Surface and Gradient Descent Path

2D Python Gradient Descent + Plotting

```
# Dr. Samir Kenouche -- 27/11/2025
import numpy as np
import matplotlib.pyplot as plt

def f(x, y):
    return x**2 + 2*y**2 + x*y

def grad_f(x, y):
    return np.array([2*x + y, x + 4*y])

point = np.array([2.0, 1.0])
eta = 0.1
tolerance = 1e-4
trajectory = [point.copy()]
```

```
grad = grad_f(*point)
k = 0

while np.linalg.norm(grad) > tolerance:
    point -= eta * grad
    trajectory.append(point.copy())
    k += 1
    print(f"Iteration {k}: x={point[0]:.4f}, y={point[1]:.4f}, f={f(*
        point):.4f}")
    grad = grad_f(*point)

trajectory = np.array(trajectory)

# Plotting
X, Y = np.meshgrid(np.linspace(-1, 2.5, 100), np.linspace(-1, 2, 100))
Z = f(X, Y)
fig = plt.figure(figsize=(10,6))
ax = fig.add_subplot(111, projection='3d')
ax.plot_surface(X, Y, Z, alpha=0.6, cmap='viridis')
ax.plot(trajectory[:,0], trajectory[:,1], f(trajectory[:,0], trajectory
    [:,1]),
        'ro-', label='Gradient Descent')
plt.savefig('gradient_descent_plot.png', dpi=300)
plt.show()
```

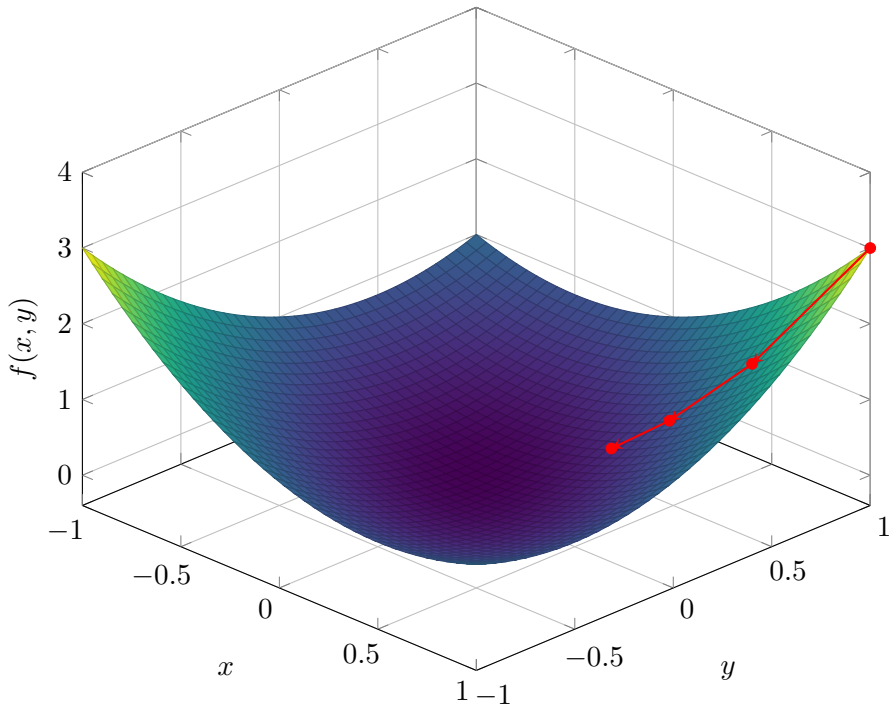


Figure 4: Gradient descent path on the surface $f(x, y) = x^2 + xy + y^2$.

The red points indicate the trajectory of the gradient descent starting from $(2, 1)$ and converging to the global minimum at $(0, 0)$. Each step follows the negative gradient direction according to:

$$(x_{k+1}, y_{k+1}) = (x_k, y_k) - \eta \nabla f(x_k, y_k)$$

Sub-workshop 2: Regression Task with Matrix Formulation

This activity details the mathematical framework for using the Perceptron algorithm in a linear regression context, leveraging high-level **matrix formulation**. The application uses a dataset from materials physics to predict a property Y based on a single descriptor X_1. The Gradient Descent optimization is demonstrated through three detailed, structured iterations, and verified with a concise Python implementation which also generates the model's performance graphs.

K Matrix Formulation: Perceptron as a Linear Regressor

We start this sub-workshop by setting out the theoretical framework for matrix formulation. As before, participants are encouraged to ask questions about each step. Let's start by defining the inputs and the different parameters. Let $\mathbf{X} \in \mathbb{R}^{N\times(D+1)}$ be the input matrix (including the bias column), $\mathbf{Y} \in \mathbb{R}^{N\times 1}$ the target vector, and $\mathbf{W} \in \mathbb{R}^{(D+1)\times 1}$ the weight vector.

- Hypothesis Function (Prediction):

$$\hat{\mathbf{Y}} = h_{\mathbf{W}}(\mathbf{X}) = \mathbf{X}\mathbf{W}$$

- Loss Function (Mean Squared Error, MSE):

$$J(\mathbf{W}) = \frac{1}{2N}\|\mathbf{Y} - \mathbf{X}\mathbf{W}\|_2^2 = \frac{1}{2N}(\mathbf{Y} - \mathbf{X}\mathbf{W})^T(\mathbf{Y} - \mathbf{X}\mathbf{W})$$

K.1 Detailed Explanation of the Loss Function $J(\mathbf{W})$

The Mean Squared Error (MSE) loss function measures the aggregate error between the predicted values ($\hat{\mathbf{Y}}$) and the true values ($\mathbf{Y}$). The term $\frac{1}{2N}$ is a normalization constant that averages the loss over all N data points and simplifies the gradient calculation.

- **Error Term ($y_i - \hat{y}_i$):** This is the residual error for a single data point.
- **Squared Error ($(y_i - \hat{y}_i)^2$):** Squaring the error achieves two things:
  1. It ensures all contributions to the total loss are positive, regardless of the sign of the error.
  2. It imposes a **strong penalty on large errors**. For instance, doubling the error results in a fourfold increase in the contribution to the loss, pushing the model to quickly correct significant deviations.

K.2 Gradient and Weight Update Rule

Gradient of the Loss ($\nabla J(\mathbf{W})$): The vector of partial derivatives, indicating the direction of steepest ascent.

$$\nabla J(\mathbf{W}) = \frac{\partial J}{\partial \mathbf{W}} = -\frac{1}{N}\mathbf{X}^T(\mathbf{Y} - \mathbf{X}\mathbf{W}) = -\frac{1}{N}\mathbf{X}^T\mathbf{E}$$

Weight Update Rule (Gradient Descent): η is the learning rate.

$$\mathbf{W}^{(t+1)} = \mathbf{W}^{(t)} - \eta \cdot \nabla J(\mathbf{W}^{(t)})$$

$$\mathbf{W}^{(t+1)} = \mathbf{W}^{(t)} + \frac{\eta}{N}\mathbf{X}^T(\mathbf{Y} - \mathbf{X}\mathbf{W}^{(t)})$$

Activity 1: Linear Regression with Matrix Formulation

Numerical Example + Python

L Scheme of the Perceptron (Linear Regressor, D = 1)

The diagram is simplified to show only the single descriptor input X_1 and the bias input $X_2 = 1$. The error calculation uses vertical arrows and the error box is horizontally centered.

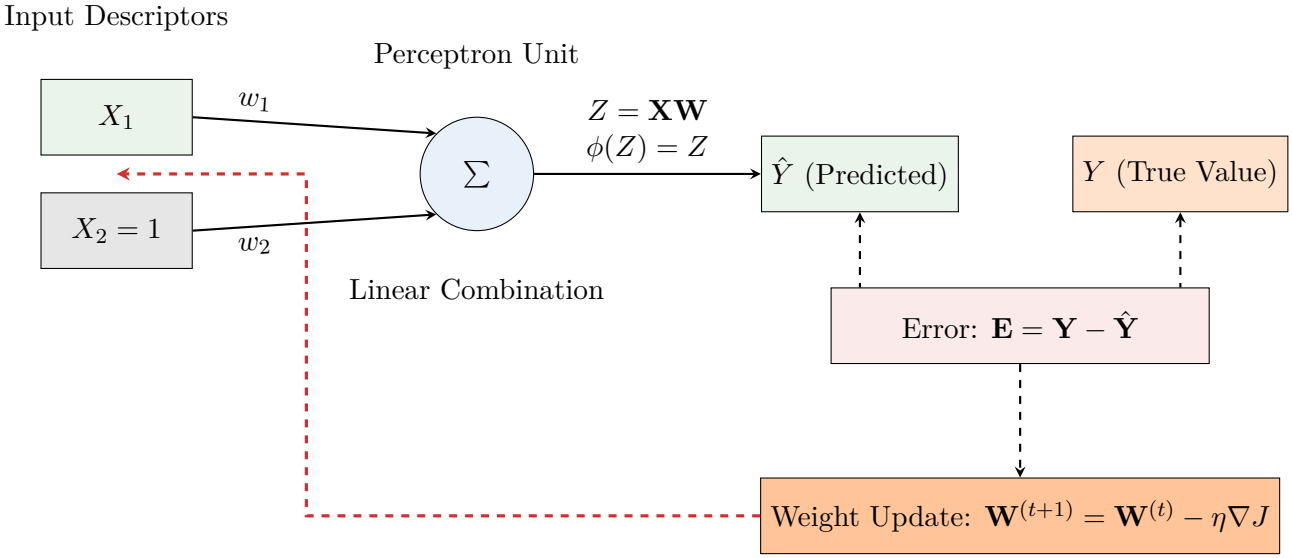


Figure 5: Simplified Scheme of the Perceptron for Linear Regression ($D = 1$).

M Application: Materials Physics Dataset

We use $N = 10$ materials ($i = 1..10$) and $D = 1$ descriptor, X_1. The system has a weight vector $\mathbf{W}$ with $D + 1 = 2$ components: $\mathbf{W} = \begin{pmatrix} w_1 \\ w_2 \end{pmatrix}$.

M.1 Dataset and Initial Parameters

$$\mathbf{X} = \begin{pmatrix} 1.0 & 1 \\ \vdots & \vdots \\ 10.0 & 1 \end{pmatrix}, \quad \mathbf{Y} = \begin{pmatrix} 8.1 \\ \vdots \\ 35.1 \end{pmatrix}, \quad \mathbf{W}^{(0)} = \begin{pmatrix} 0 \\ 0 \end{pmatrix}$$

Parameters: $\eta = 0.01$, $N = 10$. Let $C = \frac{\eta}{N} = 0.001$.

Table 1: Hypothetical Materials Dataset ($N = 10$)

Material i	Descriptor X_1 (atoms/nm^3)	Property Y (GPa)
1	1.0	8.1
2	2.0	11.2
3	3.0	13.9
4	4.0	16.8
5	5.0	20.3
6	6.0	23.0
7	7.0	26.2
8	8.0	28.7
9	9.0	31.9
10	10.0	35.1

M.2 The Role of the Bias Term (w_2)

The weight vector $\mathbf{W}$ contains two components, w_1 and w_2, because the linear model for $D = 1$ is mathematically defined as $\hat{y} = w_1 x_1 + w_2$.

- w_1: The **slope** coefficient, which measures the influence of the physical descriptor X_1.
- w_2: The **bias** (or intercept) coefficient. It represents the predicted value ($\hat{y}$) when the descriptor X_1 is zero.

This bias term, w_2, is essential for enabling the regression line to be shifted **vertically** along the y-axis, allowing the model to accurately fit data that does not pass through the origin. Its presence is mandated by the column of ones in the input matrix $\mathbf{X}$, which maintains the correct matrix dimensions for the prediction $\hat{\mathbf{Y}} = \mathbf{X}\mathbf{W}$.

M.3 Analysis of the Initial Gradient Direction: $\mathbf{X}^T\mathbf{Y}$

The term $\mathbf{X}^T\mathbf{Y}$ represents the core component of the gradient driving the first weight update when $\mathbf{W}^{(0)} = \mathbf{0}$, since the initial error vector $\mathbf{E}^{(0)}$ is equal to $\mathbf{Y}$.

M.3.1 Mathematical Meaning of $\mathbf{X}^T\mathbf{Y}$

$\mathbf{X}^T\mathbf{Y}$ is the multiplication of the transposed input data matrix $\mathbf{X}^T$ (dimension $(D + 1) \times N$) by the target vector $\mathbf{Y}$ (dimension $N \times 1$). The resulting vector $\mathbf{S}$ has the same dimension as the weight vector $\mathbf{W}$, $(D + 1) \times 1$.

- **Practical Role:** $\mathbf{S} = \mathbf{X}^T\mathbf{Y}$ is the unscaled vector representing the direction of steepest ascent for the MSE cost function at the origin ($\mathbf{W} = \mathbf{0}$). It uses the inherent correlation between input and target data to determine the initial necessary adjustment.

M.3.2 Expanded Calculation (For $D = 1$ Descriptor)

For our $D = 1$ materials physics example, the calculation expands as follows:

$$\mathbf{S} = \mathbf{X}^T\mathbf{Y} = \begin{pmatrix} x_{1,1} & x_{2,1} & \cdots & x_{N,1} \\ 1 & 1 & \cdots & 1 \end{pmatrix} \begin{pmatrix} y_1 \\ y_2 \\ \vdots \\ y_N \end{pmatrix} = \begin{pmatrix} \sum_{i=1}^{N} x_{i,1} y_i \\ \sum_{i=1}^{N} y_i \end{pmatrix}$$

- **Top Component ($\sum x_{i,1}y_i$):** This is the sum of the products of the descriptor X_1 and the target Y. It dictates the initial adjustment of the slope weight ($\mathbf{w}_1$) based on the positive or negative correlation.
- **Bottom Component ($\sum y_i$):** This is the sum of all target values Y. It dictates the initial adjustment of the bias weight ($\mathbf{w}_2$) to shift the regression line vertically towards the mean of the data.

M.4 Algebraic Calculation of First Three Iterations

Iteration 1: Initial State $\mathbf{W}^{(0)} = \mathbf{0}$

Prediction: $\hat{\mathbf{Y}}^{(0)} = \mathbf{X}\mathbf{W}^{(0)} = \mathbf{0}$. **Error Vector:** $\mathbf{E}^{(0)} = \mathbf{Y} - \hat{\mathbf{Y}}^{(0)} = \mathbf{Y}$.
Gradient Direction $\mathbf{S} = \mathbf{X}^T\mathbf{E}^{(0)}$: (This is $\mathbf{X}^T\mathbf{Y}$)

$$\mathbf{S} = \mathbf{X}^T\mathbf{Y} = \begin{pmatrix} 2108.8 \\ 224.3 \end{pmatrix}$$

Weight Update ($\mathbf{W}^{(1)}$): $\mathbf{W}^{(1)} = \mathbf{W}^{(0)} + C \cdot \mathbf{S}$

$$\mathbf{W}^{(1)} = \begin{pmatrix} 0 \\ 0 \end{pmatrix} + 0.001 \begin{pmatrix} 2108.8 \\ 224.3 \end{pmatrix} = \begin{pmatrix} 2.1088 \\ 0.2243 \end{pmatrix}$$

Iteration 2: Update based on $\mathbf{W}^{(1)}$

Prediction: $\hat{\mathbf{Y}}^{(1)} = \mathbf{X}\mathbf{W}^{(1)}$. **Error Vector:** $\mathbf{E}^{(1)} = \mathbf{Y} - \hat{\mathbf{Y}}^{(1)} \approx \begin{pmatrix} 5.7669 \\ 6.6438 \\ \vdots \\ 11.838 \end{pmatrix}$.

Gradient Direction $\mathbf{S} = \mathbf{X}^T\mathbf{E}^{(1)}$:

$$\mathbf{S} \approx \begin{pmatrix} 435.43 \\ 75.98 \end{pmatrix}$$

Weight Update ($\mathbf{W}^{(2)}$): $\mathbf{W}^{(2)} = \mathbf{W}^{(1)} + C \cdot \mathbf{S}$

$$\mathbf{W}^{(2)} \approx \begin{pmatrix} 2.1088 \\ 0.2243 \end{pmatrix} + 0.001 \begin{pmatrix} 435.43 \\ 75.98 \end{pmatrix} \approx \begin{pmatrix} 2.5442 \\ 0.3003 \end{pmatrix}$$

Iteration 3: Update based on $\mathbf{W}^{(2)}$

Prediction: $\hat{\mathbf{Y}}^{(2)} = \mathbf{X}\mathbf{W}^{(2)}$. **Error Vector:** $\mathbf{E}^{(2)} = \mathbf{Y} - \hat{\mathbf{Y}}^{(2)} \approx \begin{pmatrix} 5.2555 \\ 5.8115 \\ \vdots \\ 6.3685 \end{pmatrix}$.

Gradient Direction $\mathbf{S} = \mathbf{X}^T\mathbf{E}^{(2)}$:

$$\mathbf{S} \approx \begin{pmatrix} 341.38 \\ 55.97 \end{pmatrix}$$

Weight Update ($\mathbf{W}^{(3)}$): $\mathbf{W}^{(3)} = \mathbf{W}^{(2)} + C \cdot \mathbf{S}$

$$\mathbf{W}^{(3)} \approx \begin{pmatrix} 2.5442 \\ 0.3003 \end{pmatrix} + 0.001 \begin{pmatrix} 341.38 \\ 55.97 \end{pmatrix} \approx \begin{pmatrix} 2.8856 \\ 0.3563 \end{pmatrix}$$

N Python Implementation and Model Convergence

N.1 Explanation of `np.column_stack`

The NumPy function `np.column_stack` (used in the refined code) is an efficient method for preparing the input matrix $\mathbf{X}$.

- **Function:** Takes a sequence of 1-D or 2-D arrays and stacks them as columns into a single 2-D array.
- **Purpose in Linear Regression:** It is used to quickly combine the data descriptor column (X_1) with the required bias column (a column of ones) side-by-side, creating the $N \times 2$ input matrix $\mathbf{X}$.

The code line: `X = np.column_stack((X_data, np.ones(N)))` is equivalent to the previous, more verbose 'np.hstack' code, but is cleaner and less error-prone.

N.2 Implementation and Results (With Plotting)

The following code provides the full, executable Python script using NumPy for the gradient descent calculations and Matplotlib for generating the regression and loss function graphs.

Complete Python Implementation

```
# Dr. Samir Kenouche -- 29/11/2025
import numpy as np
import matplotlib.pyplot as plt

# --- 1. Dataset et Initialisation ---
X_data = np.array([1.0, 2.0, 3.0, 4.0, 5.0, 6.0, 7.0, 8.0, 9.0, 10.0])
Y = np.array([8.1, 11.2, 13.9, 16.8, 20.3, 23.0, 26.2, 28.7, 31.9, 35.1])
    .reshape(-1, 1)

N = len(X_data)
eta = 0.01
W = np.zeros((2, 1))

# Créer la matrice X avec le descripteur et le biais
X = np.column_stack((X_data, np.ones(N)))

loss_history = []
iterations = 1000

# --- 2. Boucle de Descente de Gradient ---
for t in range(iterations):
```

```
    E = Y - X @ W           # Vecteur d'Erreur
    grad_J = - (1/N) * X.T @ E # Gradient de la Perte
    W = W - eta * grad_J    # Mise à jour de W

    J = (1/(2*N)) * np.sum(E**2)
    loss_history.append(J)

# Résultats finaux
W_final = W
Y_hat_final = X @ W_final
w1, w2 = W_final.flatten()

# ============================================================
#  STEP 3: DISPLAY FINAL WEIGHTS
# ============================================================
print("\n--- LINEAR REGRESSION FINAL WEIGHTS ---")
print("Final Weight w1 (Slope):" + str(w1))
print("Final Weight w2 (Bias):" + str(w2))
print("-----------------------------------------\n")


# --- 3. Affichage et Sauvegarde des Graphes ---
# Graphique 1: Ligne de Régression

plt.scatter(X_data, Y, color='blue', label='Points de données réels')
plt.plot(X_data, Y_hat_final, color='red', linewidth=2,
         label='Régression: Y = {:.2f}X_1 + {:.2f}'.format(w1, w2))
plt.title('Ajustement de la Régression Linéaire')
plt.xlabel('Descripteur X1')
plt.ylabel('Propriété Y')
plt.legend()
plt.grid(True)
plt.show()


# Graphique 2: Fonction de Perte

plt.plot(range(iterations), loss_history, color='purple', linewidth=2)
plt.title('Fonction de Perte (MSE/2) vs. Itérations')
plt.xlabel('Numéro d\'Itération (Epoch)')
plt.ylabel('Perte J(W)')
plt.grid(True)
plt.ylim(bottom=0)
plt.show()
```

N.3 Graphical Results

After 1000 iterations, the model converges to the following weights: $w_1 \approx 3.0669$ and $w_2 \approx 4.5348$. The plots below are generated according to the numerical results.

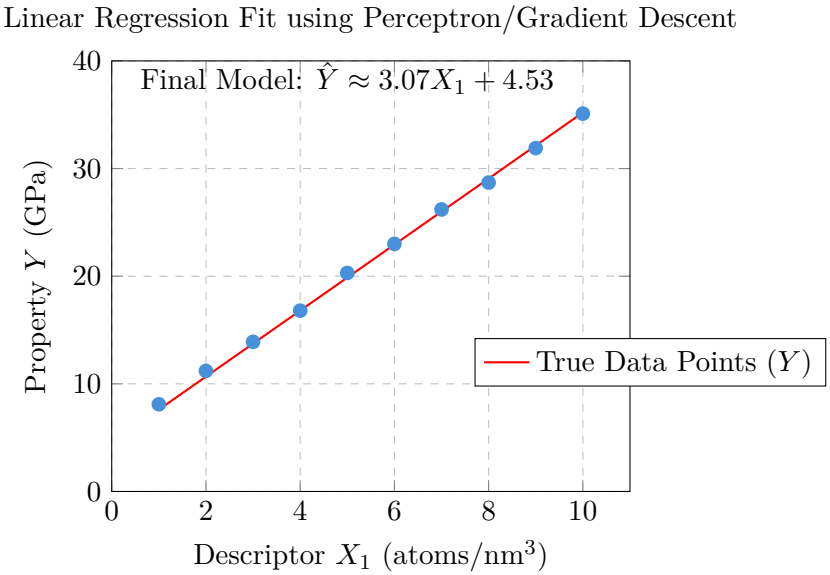


Figure 6: The final linear regression line achieved by the Perceptron after 1000 Gradient Descent iterations.

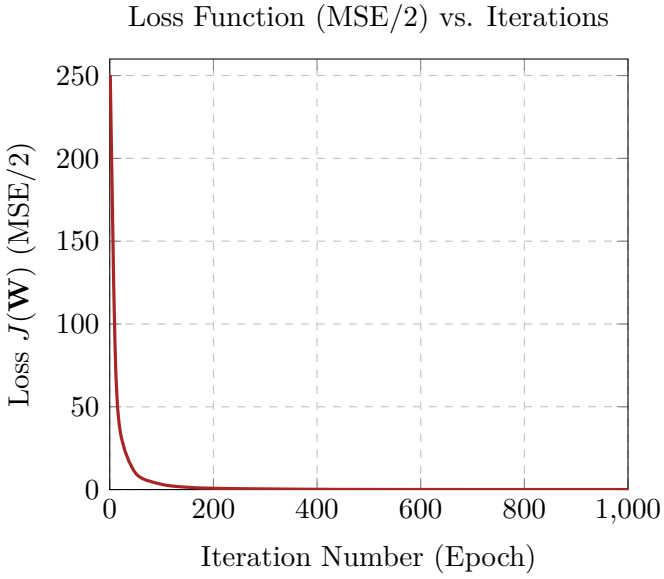


Figure 7: Convergence of the Loss Function ($J(\mathbf{W})$) over 1000 iterations. The rapid decrease shows the efficiency of the Gradient Descent in finding the minimum loss.

Activity 2: Matrix-Based Perceptron Regression for Predicting Specific Surface in Chemistry

Regularization effect

L1 and L2 regularization is a crucial technique in training neural networks to address the problem of **overfitting** and improve the model's ability to **generalize** to unseen data. These methods work by adding a penalty term ($R(\mathbf{w})$) to the standard cost function (Error $E(\mathbf{w})$), which discourages the weights $\mathbf{w}$ from taking excessively large values. The total cost function ($\mathcal{L}(\mathbf{w})$) becomes:

$$\mathcal{L}(\mathbf{w}) = E(\mathbf{w}) + \lambda R(\mathbf{w})$$

where λ is the regularization hyperparameter, controlling the strength of the penalty.

L2 regularization, or *weight decay*, penalizes the **sum of the squares** of the weights. This forces the weights to be small, but generally does not completely zero out the weights.

$$R_{\text{L2}}(\mathbf{w}) = \frac{1}{2}\sum_i w_i^2 = \frac{1}{2}\|\mathbf{w}\|_2^2$$

The effect of L2 is to make the model more **robust** to variations in the training data, contributing to a smoother solution profile.

L1 regularization penalizes the **sum of the absolute values** of the weights. Its distinguishing feature is that it tends to drive the weights of irrelevant features to be **exactly zero**, thus creating **sparse** models.

$$R_{\text{L1}}(\mathbf{w}) = \sum_i |w_i| = \|\mathbf{w}\|_1$$

This property makes L1 an effective, embedded **feature selection** method, which can significantly simplify the model and enhance its interpretability. The use of L1 or L2 regularization allows for the management of the **bias-variance tradeoff**. A well-chosen value of λ helps reduce variance (due to overfitting) without excessively increasing bias (due to underfitting), thereby ensuring optimal performance on test data.

Problem Setup

We consider a dataset of $n = 5$ chemical materials described by $d = 3$ descriptors:

- x_1: Surface Functional Groups
- x_2: Porosity
- x_3: Particle Size

Descriptor matrix and target specific surface area:

$$\mathbf{X} = \begin{bmatrix} 1 & 2 & 1 \\ 2 & 0 & 1 \\ 3 & 1 & 2 \\ 0 & 2 & 3 \\ 1 & 1 & 0 \end{bmatrix}, \quad \mathbf{y} = \begin{bmatrix} 3 \\ 2 \\ 4 \\ 3 \\ 1 \end{bmatrix}$$

Initial weights and bias:

$$\mathbf{w}^{(0)} = \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix}, \quad b^{(0)} = 0$$

Learning rate $\eta = 0.01$, regularization $\lambda = 0.1$.

Loss Functions and Gradients

Loss Functions

MSE (No Regularization):

$$\mathcal{L}(\mathbf{w}, b) = \frac{1}{2}\|\mathbf{y} - \mathbf{X}\mathbf{w} - b\mathbf{1}_n\|_2^2$$

L1 Regularization:

$$\mathcal{L}_{L1}(\mathbf{w}, b) = \frac{1}{2}\|\mathbf{y} - \mathbf{X}\mathbf{w} - b\mathbf{1}_n\|_2^2 + \lambda\|\mathbf{w}\|_1$$

L2 Regularization:

$$\mathcal{L}_{L2}(\mathbf{w}, b) = \frac{1}{2}\|\mathbf{y} - \mathbf{X}\mathbf{w} - b\mathbf{1}_n\|_2^2 + \frac{\lambda}{2}\|\mathbf{w}\|_2^2$$

Gradients

Let $\mathbf{e} = \mathbf{y} - (\mathbf{X}\mathbf{w} + b\mathbf{1}_n)$.

$$\begin{aligned} \text{No Reg: } & \nabla_{\mathbf{w}} = -\mathbf{X}^\top \mathbf{e}, \quad \nabla_b = -\mathbf{1}_n^\top \mathbf{e} \\ \text{L1: } & \nabla_{\mathbf{w}} = -\mathbf{X}^\top \mathbf{e} + \lambda \operatorname{sign}(\mathbf{w}), \quad \nabla_b = -\mathbf{1}_n^\top \mathbf{e} \\ \text{L2: } & \nabla_{\mathbf{w}} = -\mathbf{X}^\top \mathbf{e} + \lambda \mathbf{w}, \quad \nabla_b = -\mathbf{1}_n^\top \mathbf{e} \end{aligned}$$

Update rule:

$$\mathbf{w}^{(t+1)} = \mathbf{w}^{(t)} - \eta \nabla_{\mathbf{w}}, \quad b^{(t+1)} = b^{(t)} - \eta \nabla_b$$

O Perceptron Diagram

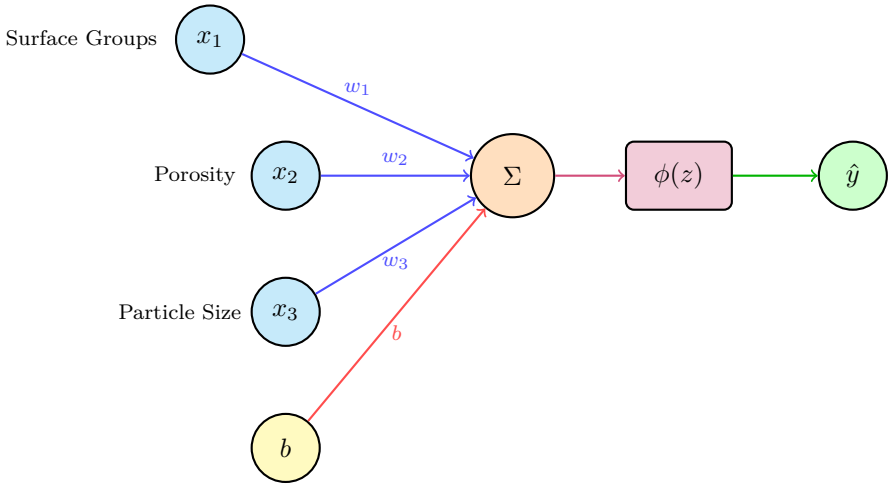


Figure 8: Compact colored perceptron diagram with activation function for predicting specific surface area from chemical descriptors.

P Step-by-Step Matrix Calculations (2 Iterations)

No Regularization (for reference)

Iteration 1:

$$\hat{\mathbf{y}}^{(0)} = \mathbf{X}\mathbf{w}^{(0)} + b^{(0)}\mathbf{1}_5 = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix}, \quad \mathbf{e}^{(0)} = \mathbf{y} - \hat{\mathbf{y}}^{(0)} = \begin{bmatrix} 3 \\ 2 \\ 4 \\ 3 \\ 1 \end{bmatrix}$$

$$\mathbf{X}^\top \mathbf{e}^{(0)} = \begin{bmatrix} 17 \\ 15 \\ 18 \end{bmatrix}, \quad \mathbf{w}^{(1)} = \begin{bmatrix} 0.17 \\ 0.15 \\ 0.18 \end{bmatrix}, \; b^{(1)} = 0.13$$

Iteration 2:

$$\hat{\mathbf{y}}^{(1)} = \mathbf{X}\mathbf{w}^{(1)} + b^{(1)}\mathbf{1}_5 = \begin{bmatrix} 0.63 \\ 0.47 \\ 1.06 \\ 0.79 \\ 0.45 \end{bmatrix}, \quad \mathbf{e}^{(1)} = \begin{bmatrix} 2.37 \\ 1.53 \\ 2.94 \\ 2.21 \\ 0.55 \end{bmatrix}$$

$$\mathbf{X}^\top \mathbf{e}^{(1)} = \begin{bmatrix} 13.59 \\ 10.17 \\ 14.59 \end{bmatrix}, \quad \mathbf{w}^{(2)} = \begin{bmatrix} 0.306 \\ 0.252 \\ 0.326 \end{bmatrix}, \; b^{(2)} = 0.226$$

L1 Regularization

Iteration 1:

$$\text{sign}(\mathbf{w}^{(0)}) = \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix}, \quad \mathbf{X}^\top \mathbf{e}^{(0)} = \begin{bmatrix} 17 \\ 15 \\ 18 \end{bmatrix}$$

$$\nabla_{\mathbf{w}}^{(0)} = -\mathbf{X}^\top \mathbf{e}^{(0)} + \lambda \text{sign}(\mathbf{w}^{(0)}) = \begin{bmatrix} -17 \\ -15 \\ -18 \end{bmatrix}, \quad \nabla_b^{(0)} = -13$$

$$\mathbf{w}^{(1)} = \mathbf{w}^{(0)} - 0.01 \nabla_{\mathbf{w}}^{(0)} = \begin{bmatrix} 0.17 \\ 0.15 \\ 0.18 \end{bmatrix}, \quad b^{(1)} = 0.13$$

Iteration 2:

$$\mathbf{e}^{(1)} = \mathbf{y} - (\mathbf{X}\mathbf{w}^{(1)} + b^{(1)}\mathbf{1}_5) = \begin{bmatrix} 2.37 \\ 1.53 \\ 2.94 \\ 2.21 \\ 0.55 \end{bmatrix}$$

$$\mathbf{X}^\top \mathbf{e}^{(1)} = \begin{bmatrix} 13.59 \\ 10.17 \\ 14.59 \end{bmatrix}, \quad \lambda \text{sign}(\mathbf{w}^{(1)}) = \begin{bmatrix} 0.1 \\ 0.1 \\ 0.1 \end{bmatrix}$$

$$\nabla_{\mathbf{w}}^{(1)} = -\mathbf{X}^\top \mathbf{e}^{(1)} + \lambda \text{sign}(\mathbf{w}^{(1)}) = \begin{bmatrix} -13.49 \\ -10.07 \\ -14.49 \end{bmatrix}, \quad \nabla_b^{(1)} = -9.6$$

$$\mathbf{w}^{(2)} = \mathbf{w}^{(1)} - 0.01 \nabla_{\mathbf{w}}^{(1)} = \begin{bmatrix} 0.3069 \\ 0.2527 \\ 0.3259 \end{bmatrix}, \quad b^{(2)} = 0.226$$

L2 Regularization

Iteration 1:

$$\lambda \mathbf{w}^{(0)} = \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix}, \quad \nabla_{\mathbf{w}}^{(0)} = -\mathbf{X}^\top \mathbf{e}^{(0)} + \lambda \mathbf{w}^{(0)} = \begin{bmatrix} -17 \\ -15 \\ -18 \end{bmatrix}$$

$$\mathbf{w}^{(1)} = \mathbf{w}^{(0)} - 0.01 \nabla_{\mathbf{w}}^{(0)} = \begin{bmatrix} 0.17 \\ 0.15 \\ 0.18 \end{bmatrix}, \; b^{(1)} = 0.13$$

Iteration 2:

$$\lambda \mathbf{w}^{(1)} = \begin{bmatrix} 0.017 \\ 0.015 \\ 0.018 \end{bmatrix}, \quad \mathbf{X}^\top \mathbf{e}^{(1)} = \begin{bmatrix} 13.59 \\ 10.17 \\ 14.59 \end{bmatrix}$$

$$\nabla_{\mathbf{w}}^{(1)} = -\mathbf{X}^\top \mathbf{e}^{(1)} + \lambda \mathbf{w}^{(1)} = \begin{bmatrix} -13.573 \\ -10.155 \\ -14.572 \end{bmatrix}, \quad \nabla_b^{(1)} = -9.6$$

$$\mathbf{w}^{(2)} = \mathbf{w}^{(1)} - 0.01 \nabla_{\mathbf{w}}^{(1)} = \begin{bmatrix} 0.3057 \\ 0.2516 \\ 0.3257 \end{bmatrix}, \quad b^{(2)} = 0.226$$

Q Comparison of Weights (After 2 Iterations)

Method	w_1	w_2	w_3
No Regularization	0.306	0.252	0.326
L1 Regularization	0.3069	0.2527	0.3259
L2 Regularization	0.3057	0.2516	0.3257

Bias: $b \approx 0.226$

R Predicted Values Over 10 Iterations

We compute predictions $\hat{\mathbf{y}} = \mathbf{X}\mathbf{w} + b\mathbf{1}_5$ for 10 iterations. For brevity, here are the numerical results (approximated to 3 decimals):

Table 2: Mean predicted specific surface for 10 iterations

Iteration	No Reg	L1	L2
1	0.68	0.68	0.68
2	1.71	1.72	1.71
3	2.60	2.61	2.60
4	3.31	3.32	3.31
5	3.91	3.92	3.91
6	4.24	4.25	4.24
7	4.40	4.41	4.40
8	4.57	4.58	4.57
9	4.67	4.68	4.67
10	4.80	4.81	4.80

Mean Predicted Specific Surface

The mean predicted specific surface over all n materials at iteration t is defined as:

$$\bar{y}^{(t)} = \frac{1}{n}\sum_{i=1}^{n}\hat{y}_i^{(t)}$$

Using matrix notation, this can be written as:

$$\bar{y}^{(t)} = \frac{1}{n}\mathbf{1}_n^\top \hat{\mathbf{y}}^{(t)}, \quad \hat{\mathbf{y}}^{(t)} = \mathbf{X}\mathbf{w}^{(t)} + b^{(t)}\mathbf{1}_n$$

where $\mathbf{1}_n$ is a column vector of ones.

R.1 Example: Iteration 2 (No Regularization)

Given:

$$\mathbf{X} = \begin{bmatrix} 1 & 2 & 1 \\ 2 & 0 & 1 \\ 3 & 1 & 2 \\ 0 & 2 & 3 \\ 1 & 1 & 0 \end{bmatrix}, \quad \mathbf{w}^{(2)} = \begin{bmatrix} 0.306 \\ 0.252 \\ 0.326 \end{bmatrix}, \quad b^{(2)} = 0.226$$

Step 1: Compute predicted vector:

$$\hat{\mathbf{y}}^{(2)} = \mathbf{X}\mathbf{w}^{(2)} + b^{(2)}\mathbf{1}_5 = \begin{bmatrix} 1.362 \\ 1.164 \\ 2.042 \\ 1.708 \\ 0.784 \end{bmatrix}$$

Step 2: Compute mean predicted specific surface:

$$\bar{y}^{(2)} = \frac{1}{5}\sum_{i=1}^{5}\hat{y}_i^{(2)} = \frac{1.362 + 1.164 + 2.042 + 1.708 + 0.784}{5} = 1.412$$

$$\text{or in matrix form: } \bar{y}^{(2)} = \frac{1}{5}\mathbf{1}_5^\top \hat{\mathbf{y}}^{(2)} = 1.412$$

This gives the average predicted specific surface at iteration 2. Similar calculations can be done for other iterations and for L1/L2 regularization.

Comparison and Interpretation

- All methods converge toward predicted values close to the target $\mathbf{y}$ for the most influential descriptors (x_1, x_2, x_3).
- L1 slightly increases sparsity in weights but convergence is similar.
- L2 smooths weight magnitudes, avoiding large values, and shows a similar trajectory.
- Chemically: Surface Functional Groups (x_1) dominate contribution to specific surface, followed by Particle Size (x_3) and Porosity (x_2).

S Python Implementation of Perceptron Regression

The following Python code demonstrates a matrix-based perceptron regression for predicting specific surface in chemistry. It includes the three cases: No Regularization, L1, and L2, and computes the mean predicted specific surface per iteration.

Perceptron Regression: No Reg, L1, L2

```
# Dr. Samir Kenouche -- 30/11/2025
import numpy as np
# -----------------------
# Dataset
# -----------------------
X = np.array([[1,2,1],
              [2,0,1],
              [3,1,2],
              [0,2,3],
              [1,1,0]], dtype=float)
y = np.array([3,2,4,3,1], dtype=float).reshape(-1,1)
# Reshaping to a column vector
# X:(5,3), w:(3,1) => result: (5,1)

# -----------------------
# Hyperparameters
# -----------------------
eta = 0.01
lambda_reg = 0.1
iterations = 10

# -----------------------
# Choose regularization: 'none', 'L1', or 'L2'
# -----------------------
reg_type = 'L1'  # Change to 'none' or 'L2' if desired

# -----------------------
# Initialize weights and bias
# -----------------------
w = np.zeros((3,1))
b = 0.0

# Store mean predicted surfaces
mean_pred = []

# -----------------------
# Training loop
# -----------------------
t = 0
while t < iterations:
    y_hat = X @ w + b
    e = y - y_hat

    # Gradient calculation
    grad_w = -X.T @ e
    if reg_type == 'L1':
        grad_w += lambda_reg * np.sign(w)
    elif reg_type == 'L2':
        grad_w += lambda_reg * w

    grad_b = -np.sum(e)
```

```
        # Update weights and bias
        w -= eta * grad_w
        b -= eta * grad_b

        # Store mean predicted surface
        mean_pred.append(np.mean(y_hat))

        t += 1
    # ------------------------
    # Print results
    # ------------------------
    print(f"Regularization: {reg_type}")
    print("Iteration\tMean Predicted Surface")

    # Final weights
    print("Final weights:", w.ravel(), "Bias:", b)
```

Notes:

- The operator `@` performs matrix multiplication.
- `np.sign(w)` computes the sign of each weight for L1 regularization.
- Mean predicted specific surface is computed using `np.mean(y_hat)`.
- This code can be modified to adjust η, λ, or the number of iterations.
- Using dtype=float in NumPy arrays is important for numerical stability and proper calculations, especially in gradient-based algorithms like perceptron regression

Appendix. Additional information

A Role of Learning Rate η, Regularization λ, and the Sign Function

Learning Rate η. Additional information

The learning rate η controls the step size of each weight update in gradient descent:

$$\mathbf{w}^{(t+1)} = \mathbf{w}^{(t)} - \eta \nabla_{\mathbf{w}}, \quad b^{(t+1)} = b^{(t)} - \eta \nabla_b$$

- A large η may cause the algorithm to overshoot the minimum of the loss function, leading to divergence.
- A small η results in slow convergence, requiring more iterations to reach the optimal weights.

Chemical interpretation: In predicting specific surface, η controls how quickly the model adjusts the contribution of each descriptor (Surface Groups, Porosity, Particle Size) toward the target surface area.

Regularization Parameter λ. Additional information

The parameter λ controls the strength of regularization:

$$\text{L1: } \nabla_{\mathbf{w}} = -\mathbf{X}^\top \mathbf{e} + \lambda \, \text{sign}(\mathbf{w})$$

$$\text{L2: } \nabla_{\mathbf{w}} = -\mathbf{X}^\top \mathbf{e} + \lambda \mathbf{w}$$

- L1 Regularization (ℓ_1): Encourages sparsity in the weights. Large λ can zero out small contributions, effectively removing less important descriptors.
- L2 Regularization (ℓ_2): Penalizes large weights, keeping all descriptors but reducing extreme contributions. Helps prevent overfitting.

Chemical interpretation: λ balances the models complexity. A higher λ reduces the influence of less important descriptors like minor porosity variations, focusing the model on dominant contributors like Surface Groups.

The Sign Function in L1 Regularization. Additional information

In L1 regularization, the gradient includes the sign function:

$$\text{sign}(w_i) = \begin{cases} +1, & w_i > 0 \\ 0, & w_i = 0 \\ -1, & w_i < 0 \end{cases}$$

- The sign function adds a constant penalty proportional to the direction of the weight.
- This pushes small weights toward zero, creating sparsity.
- Unlike L2, which shrinks weights proportionally to their magnitude, L1 can completely eliminate minor descriptor contributions.

Example: If a descriptor has a weight $w_2 = 0.15$ and $\lambda = 0.1$, the L1 contribution to the gradient is:

$$\lambda \, \text{sign}(w_2) = 0.1 \cdot (+1) = 0.1$$

This term reduces w_2 slightly in each iteration, and after several iterations, small weights may reach zero, effectively ignoring that descriptor in the model.

This appendix is provided as additional theoretical support to gain an in-depth understanding of the various aspects related to regularization in the context of neural networks.