Assignment 1

Machine Learning 1, SS24

Team Members		
Last name	First name	Matriculation Number
Nožić	Naida	12336462
Hinum-Wagner	Jakob	01430617

1 Linear Regression– Detection of memristor faults

Information in the human brain is processed by *neurons* and stored in the form of *synaptic weights*. *Synaptic plasticity* — the ability to increase or decrease these weights — is the underlying mechanism responsible for knowledge-based learning. For the implementation of learning in neuro-inspired computing chips, nano-scale hardware devices, so-called *memristors*, have been proposed and fabricated. A memristor is a resistor with programmable (adjustable) resistance, hence a suitable candidate that mimics biological synapses. However, due to a number of non-idealities (e.g., fabrication, operational constraints, limited endurance), memristors, when programmed, often show deviations from the intended behavior. *Stuck memristors* do not change their resistance regardless of the magnitude of resistance change. Other memristors underestimate or overestimate the magnitude of resistance change, and we refer to such faulty behavior as *concordant faults*. Some memristors even produce resistance change in the opposite direction of the intended one, and we refer to such faults as *discordant faults*. In addition, when being programmed, the achieved resistance states are always noisy due to *programming noise*. An illustration of a memristor with an ideal behavior, a discordant fault, a stuck fault, and a concordant fault can be seen in Fig. 1A, B, C, D, respectively.

To detect memristor faults, one can use a linear regression model, interpret its parameters and finally, classify the faults. In this exercise, we will use two linear regression models, and decide which one works better.

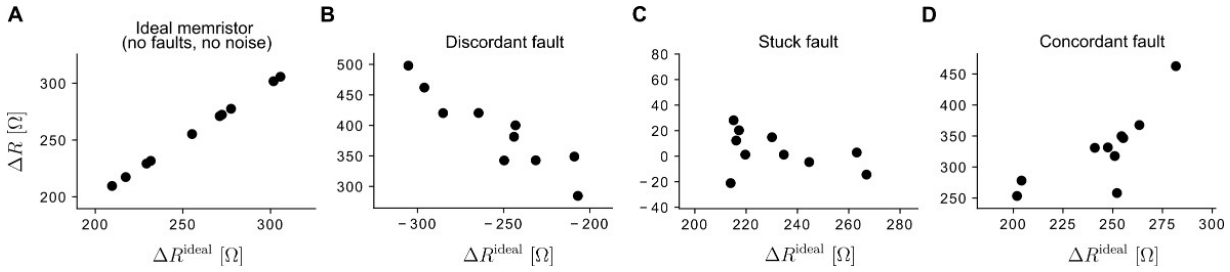


Figure 1: **Examples of memristor faults.** A memristor with (**A**) an ideal behavior (no faults, no noise), (**B**) a discordant fault, (**C**) a stuck fault, (**D**) a concordant fault. Expected resistance changes are given on the $x-$axis ($\Delta R^{\text{ideal}}[\Omega]$), the achieved ones on the $y-$axis ($\Delta R[\Omega]$).

1.1 Task 1:Problem

Model 1. Use a zero-intercept linear regression model, that is, the hypothesis $h_\theta(\Delta R^{\text{ideal}}) = \theta \cdot \Delta R^{\text{ideal}}$, with only one parameter, θ. In this case, the error function to minimize reads as follows:

$$E(\theta) = \frac{1}{m}\sum_{i=1}^{m}(h_\theta(\Delta R_i^{\text{ideal}}) - \Delta R_i)^2,$$

where m is the number of data points (measurements) we use for regression, more precisely, $(\Delta R_i^{\text{ideal}}, \Delta R_i)$, $i \in 1, ..., m$ are the points used for fitting.

We wish to find a global minimizer θ^*, i.e.,

$$\theta^* = \underset{\theta}{\operatorname{argmin}}\, E(\theta) \tag{1}$$

Derive a *closed-form analytical solution* for θ^*. (Find the derivative of the error function w.r.t. θ, set it to zero, and express θ^*. Include *all steps* of your derivation in the report.) **NOTE:** Your expression for θ^* should include sums – use the error function as given in this exercise sheet, without transformations.

Implement the equation for θ^* in the code (file `lin_reg_memristors.py`, function `fit_zero_intercept_lin_model`) by using the expressions with sums you have derived.

1.2 Task 1:Solution

Given a zero-intercept linear regression model, the regression must pass through the origin, leading to the model equation:

$$y = \theta x$$

Here, y is the dependent variable, x is the independent variable, and θ represents the slope of the line. The error function $E(\theta)$ is defined as the mean squared error between the predicted and actual values:

$$E(\theta) = \frac{1}{m}\sum_{i=1}^{m}(\theta x^{(i)} - y^{(i)})^2$$

where m is the number of observations.

To find the value of θ that minimizes the error function, differentiate $E(\theta)$ with respect to θ and set the derivative to zero:

$$\frac{dE(\theta)}{d\theta} = \frac{2}{m}\sum_{i=1}^{m}(\theta x^{(i)} - y^{(i)})x^{(i)}$$

Using the power rule and the chain rule, each term in the summation is differentiated.

Setting the derivative equal to zero for minimization:

$$\sum_{i=1}^{m}(\theta(x^{(i)})^2 - y^{(i)}x^{(i)}) = 0$$

This equation states that the weighted sum of squared inputs scaled by θ must equal the cross product of the inputs and outputs.

Isolating θ from the above equation gives:

$$\theta = \frac{\sum_{i=1}^{m}(y^{(i)}x^{(i)})}{\sum_{i=1}^{m}(x^{(i)})^2}$$

Here, the numerator represents the sum of the products of corresponding input and output values, indicating covariance between x and y. The denominator is the sum of the squares of the inputs, representing their total variance.

The obtained θ will be used in the `fit_zero_intercept_lin_model` function. It is also important to note that the error function is truly minimized for this θ, because the second derivative will always be positive.

$$E(\theta)'' = \frac{2}{m}\sum_{i=1}^{m}\theta * (x^{(\mathrm{i})})^2 - \frac{2}{m}\sum_{i=1}^{m} *(y^{(\mathrm{i})} * x^{(\mathrm{i})}) = \frac{2}{m}\sum_{i=1}^{m}(x^{(\mathrm{i})})^2$$

The second derivative does not depend on θ and is always positive. Therefore, the error function is minimized. However, if x is zero then we would not be able to find an equation that best fits the data. With having fully implemented the `fit_zero_intercept_lin_model` function, the generated plots are given at Figure 2.

The `fit_zero_intercept_lin_model` function implemented in the code computes the slope θ for a linear model without an intercept, suitable for scenarios where variables are directly proportional. It takes numpy arrays x and y, representing the x-coordinates ($\Delta R_i^{\text{ideal}}$) and y-coordinates (ΔR_i) of the data points, respectively. The slope θ is calculated using the formula:

$$\theta = \frac{\sum(x_i \cdot y_i)}{\sum(x_i^2)},$$

derived from the normal equation for linear regression when the intercept is zero. This calculation uses numpy's vectorized operations to ensure efficiency and robustness, provided x and y are of equal length and x contains no zeros.

1.3 Task 2: Problem

Model 2. Use a linear regression model with intercept, that is, the hypothesis $h_{\theta_0,\theta_1}(\Delta R^{\text{ideal}}) = \theta_0 + \theta_1 \Delta R^{\text{ideal}}$. The error function, in this case, is given by:

$$E(\theta_0, \theta_1) = \frac{1}{m} \sum_{i=1}^{m} (h_{\theta_0,\theta_1}(\Delta R_i^{\text{ideal}}) - \Delta R_i)^2,$$

where m is the number of data points (measurements) we use for regression, and $(\Delta R_i^{\text{ideal}}, \Delta R_i), i \in 1, ..., m$ the points used for fitting. Again, we seek

$$(\theta_0^*, \theta_1^*) = \underset{\theta_0,\theta_1}{\operatorname{argmin}} E(\theta_0, \theta_1) \tag{2}$$

Derive a *closed-form analytical solution* for θ_0^* and θ_1^* (which will again include sums). (Find the *partial* derivatives of the error function w.r.t. θ_0 and θ_1, set both of them to zero, and express θ_0^* and θ_1^*. Include *all steps* of your derivation in the report.) **NOTE:** Your expression for θ^* should include sums – use the error function as given in this exercise sheet, without transformations.

Implement the equations for θ_0^* and θ_1^* in the code (file `lin_reg_memristors.py`, function `fit_lin_model_with_intercept`) by using the expressions with sums you have derived.

1.4 Task 2: Solution

To derive the coefficients θ_0 and θ_1 for the linear regression model with intercept, we will follow a systematic and detailed approach by employing calculus and linear algebra. The objective is to minimize the error function, which is the sum of squared differences between observed values and values predicted by the model.

Model Definition

The model is defined as:

$$h_{\theta_0,\theta_1}(\Delta R_{\text{ideal}}) = \theta_0 + \theta_1 \Delta R_{\text{ideal}}$$

This function maps the input ΔR_{ideal} (expected change in resistance) to the output ΔR (observed change in resistance), parameterized by θ_0 (intercept) and θ_1 (slope).

The error function, representing the sum of squared residuals (differences between observed and predicted values), is:

$$E(\theta_0, \theta_1) = \frac{1}{m} \sum_{i=1}^{m} (\theta_0 + \theta_1 \Delta R_{\text{ideal},i} - \Delta R_i)^2$$

One starts by expanding the derivative:

$$\frac{\partial E}{\partial \theta_0} = \frac{\partial}{\partial \theta_0} \left(\frac{1}{m} \sum_{i=1}^{m} (\theta_0 + \theta_1 \Delta R_{\text{ideal},i} - \Delta R_i)^2 \right)$$

Using the chain rule, one has:

$$\frac{\partial E}{\partial \theta_0} = \frac{2}{m} \sum_{i=1}^{m} (\theta_0 + \theta_1 \Delta R_{\text{ideal},i} - \Delta R_i) \cdot 1$$

Set this derivative equal to zero to find the optimal θ_0:

$$\frac{2}{m} \sum_{i=1}^{m} (\theta_0 + \theta_1 \Delta R_{\text{ideal},i} - \Delta R_i) = 0$$

$$\sum_{i=1}^{m} (\theta_0 + \theta_1 \Delta R_{\text{ideal},i} - \Delta R_i) = 0$$

Similarly, taking the derivative with respect to θ_1:

$$\frac{\partial E}{\partial \theta_1} = \frac{\partial}{\partial \theta_1}\left(\frac{1}{m}\sum_{i=1}^{m}(\theta_0 + \theta_1 \Delta R_{\text{ideal},i} - \Delta R_i)^2\right)$$

Applying the chain rule, one gets:

$$\frac{\partial E}{\partial \theta_1} = \frac{2}{m}\sum_{i=1}^{m}(\theta_0 + \theta_1 \Delta R_{\text{ideal},i} - \Delta R_i) \cdot \Delta R_{\text{ideal},i}$$

Setting this to zero for optimization:

$$\frac{2}{m}\sum_{i=1}^{m}(\theta_0 + \theta_1 \Delta R_{\text{ideal},i} - \Delta R_i)\Delta R_{\text{ideal},i} = 0$$

$$\sum_{i=1}^{m}(\theta_0 + \theta_1 \Delta R_{\text{ideal},i} - \Delta R_i)\Delta R_{\text{ideal},i} = 0$$

From the equation derived from the derivative with respect to θ_0 (

$$\frac{\partial E}{\partial \theta_0} = 0$$

), one isolates θ_0:

$$m\theta_0 + \theta_1 \sum_{i=1}^{m} \Delta R_{\text{ideal},i} = \sum_{i=1}^{m} \Delta R_i$$

$$\theta_0 = \frac{\sum_{i=1}^{m} \Delta R_i - \theta_1 \sum_{i=1}^{m} \Delta R_{\text{ideal},i}}{m}$$

Plugging this expression for θ_0 into the equation derived from the derivative with respect to θ_1 ((

$$\frac{\partial E}{\partial \theta_1} = 0$$

)and simplifying by multiplying with m, one derives for θ_1:

$$\sum_{i=1}^{m}\left(\sum_{i=1}^{m}\Delta R_i - \theta_1 \sum_{i=1}^{m}\Delta R_{\text{ideal},i} + \theta_1 \Delta R_{\text{ideal},i} - \Delta R_i\right)\Delta R_{\text{ideal},i} = 0$$

$$\theta_1\left(\sum_{i=1}^{m}(\Delta R_{\text{ideal},i})^2 - \frac{(\sum_{i=1}^{m}\Delta R_{\text{ideal},i})^2}{m}\right) = \sum_{i=1}^{m}\Delta R_i \Delta R_{\text{ideal},i} - \frac{\sum_{i=1}^{m}\Delta R_i \sum_{i=1}^{m}\Delta R_{\text{ideal},i}}{m}$$

$$\theta_1 = \frac{\sum_{i=1}^{m}\Delta R_i \Delta R_{\text{ideal},i} - \frac{\sum_{i=1}^{m}\Delta R_i \sum_{i=1}^{m}\Delta R_{\text{ideal},i}}{m}}{\sum_{i=1}^{m}(\Delta R_{\text{ideal},i})^2 - \frac{(\sum_{i=1}^{m}\Delta R_{\text{ideal},i})^2}{m}}$$

Finally, substitute the value of θ_1 back into the formula for θ_0 to get:

$$\theta_0 = \frac{\sum_{i=1}^{m}\Delta R_i}{m} - \theta_1 \frac{\sum_{i=1}^{m}\Delta R_{\text{ideal},i}}{m}$$

This completes the derivation of θ_0 and θ_1, providing the coefficients necessary to minimize the error function in a linear regression model with an intercept.

The `fit_lin_model_with_intercept` function implemented in the code calculates the parameters θ_0 (intercept) and θ_1 (slope) for a linear regression model using numpy arrays x and y. These arrays represent the x-coordinates ($\Delta R_i^{\text{ideal}}$) and y-coordinates (ΔR_i), respectively, of the data points. The function employs

the least squares method to determine the best-fitting line that minimizes the sum of squared residuals between the observed values in the dataset and those predicted by the linear model.

The specific calculations are as follows: - n is the number of observations. - `sum_x` is the sum of the x values. - `sum_y` is the sum of the y values. - `sum_xy` is the sum of the product of corresponding x and y values. - `sum_x2` is the sum of the squares of x values.

The parameters are calculated using the formulas:

$$\theta_1 = \frac{n \cdot \sum_{xy} - \sum_x \cdot \sum_y}{n \cdot \sum_{x^2} - (\sum_x)^2}$$

$$\theta_0 = \frac{\sum_y - \theta_1 \cdot \sum_x}{n}$$

These parameters, θ_0 and θ_1, represent the intercept and slope of the line, respectively. The function returns these values, allowing for the construction and evaluation of a linear model that accounts for both the slope and the y-intercept.

1.5 Task 3: Problem

The data set with measurements for 8 memristors is already loaded (file `main.py`, function `task_1`). It contains a multidimensional numpy array with a shape of $(8, 10, 2)$, representing measurements for 8 memristors; for each memristor there are 10 measurements in the form $(\Delta R_i^{\text{ideal}}, \Delta R_i), i \in 1, ..., 10$, i.e., pairs representing an expected (ideal) resistance change, and an achieved resistance change.

Implement a function call to fit Model 1 to the data (per memristor), and visualize the data and the fit for each memristor. The code that generates and saves the plot is already implemented.

Implement a function call to fit Model 2 to the data (per memristor), and visualize the data and the fit for each memristor.

Include all generated plots in the report.

1.6 Task 3:Solution

The implemented Python script estimates parameters for a zero-intercept linear model for multiple memristors, using data organized where each row corresponds to a different memristor's paired data points (x, y). Initially, it determines the number of memristors by evaluating the first dimension of the data array and initializes an array to store the estimated parameters for each memristor. The script iterates through each memristor's data, extracting x and y values, fitting a zero-intercept linear model using these values, and storing the resulting slope θ into the array. After processing all data sets, it visualizes the fit results for each memristor and prints the estimated parameters. This process assumes that the data array is structured correctly and that the function $fit\_zero\_intercept\_lin\_model$ efficiently computes the linear fits without intercepts, emphasizing compact and structured data processing and visualization, as it can be seen in Figure 2

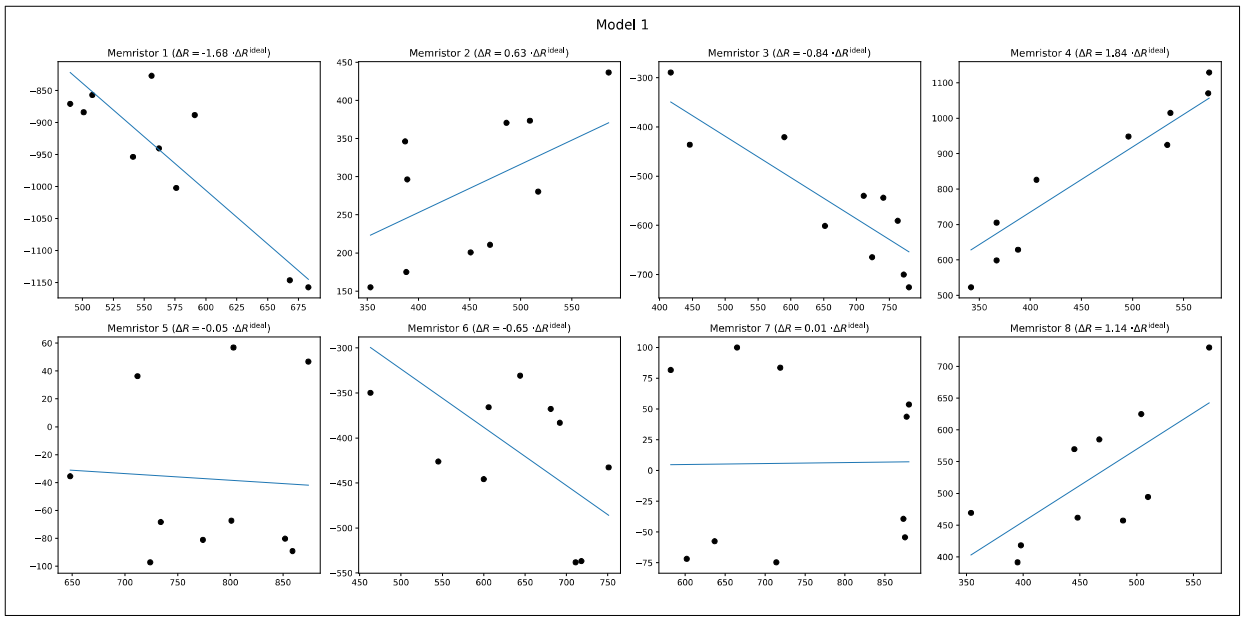


Figure 2: Plot of memristor model 1 without intercept according to Task 1 on the 8 different measured memristors

In a similar manner the code to get model two is implemented, it is initializes a zero-filled NumPy array *estimated_params_per_memristor_model2* to store two parameters (intercept and slope) for each memristor. It then iterates over each memristor, extracting x and y data from a nested data array data. For each dataset, it fits a linear regression model with an intercept using the function $fit\_lin_model\_with\_intercept(x, y)$, which returns the parameters θ_0 (intercept) and θ_1 (slope). These parameters are stored in the initialized array. After fitting models for all memristors, the *plot_model2* function is called to visualize each dataset alongside its corresponding regression line, allowing for a visual assessment of the fit, which can be seen in Figure 3

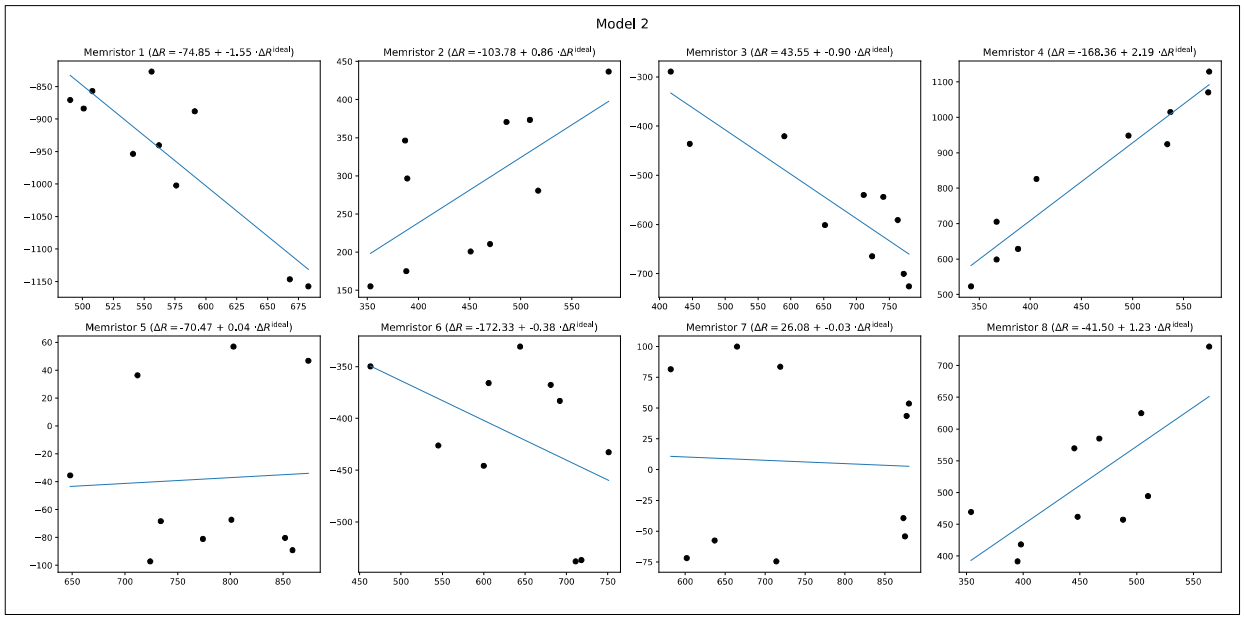


Figure 3: Plot of memristor model 2 without intercept according to Task 2 on the 8 different measured memristors

1.7 Task 4: Problem

Interpret the parameters of Model 1. Based on the parameter θ, how can we decide if there is a discordant, stuck, or concordant memristor fault? Interpret the parameters of Model 2, and state how you would use them to decide on the type of the memristor fault.

1.8 Task 4: Solution

The parameter θ in Model 1, derived through zero-intercept linear regression, acts as a scaling factor between the intended change in resistance, ΔR^{ideal}, and the actual observed change, ΔR. Understanding θ is essential for identifying potential faults in memristors. Here is how θ can be utilized to diagnose different types of memristor faults:

- $\theta = 1$: Indicates an ideal, fault-free scenario where the resistance adjusts exactly as planned, showing that ΔR directly matches ΔR^{ideal}.
- $\theta = 0$: Suggests a *stuck fault*. There is no change in resistance regardless of the intended adjustment, leading to ΔR remaining zero for all ΔR^{ideal} values.
- $0 < \theta < 1$: Points to a *concordant fault*, where the resistance change is consistently less than intended but in the correct direction. This under-response indicates that the memristor does not achieve the target resistance levels fully.
- $\theta > 1$: Also indicates a *concordant fault*, but in this scenario, the resistance change surpasses the intended levels. This over-response results in higher than expected changes in resistance.
- $\theta < 0$: Denotes a *discordant fault*. The resistance changes in the opposite direction to what was intended. For example, an expected increase in resistance results in a decrease, and vice versa.

For Model 2, which incorporates both a slope θ_1 and an intercept θ_0, the interpretation extends beyond the simple scaling observed in Model 1 to account for baseline adjustments in resistance change.

The parameters θ_0 and θ_1 in Model 2, derived through linear regression with an intercept, refine the relationship between the intended change in resistance, ΔR^{ideal}, and the actual observed change, ΔR. Understanding both θ_0 and θ_1 is essential for a comprehensive diagnosis of potential faults in memristors.

θ_1 (Slope):

- $\theta_1 = 1$: Indicates an ideal, fault-free scenario where the resistance adjusts exactly as intended, demonstrating that ΔR directly matches ΔR^{ideal}, adjusted by θ_0.
- $\theta_1 = 0$: Suggests a *stuck fault*, influenced by the intercept. The resistance change remains constant and equals θ_0 for all values of ΔR^{ideal}.
- $0 < \theta_1 < 1$: Points to a *concordant fault*, where the resistance change is proportionally less than intended but in the correct direction, adjusted by θ_0.
- $\theta_1 > 1$: Also indicates a *concordant fault*, where the resistance change surpasses the intended levels proportionally, adjusted by θ_0.
- $\theta_1 < 0$: Denotes a *discordant fault*. The resistance changes in the opposite direction to what was intended, influenced by θ_0.

θ_0 (Intercept)

- **Positive** θ_0: Suggests a baseline increase in resistance that adds a fixed amount to each intended change, possibly indicating systematic overestimations.
- **Negative** θ_0: Indicates a baseline decrease in resistance, suggesting systematic underestimations or inherent negative bias in resistance change measurements.

1.9 Task 5: Task

Which model would you prefer to use – Model 1 or Model 2? Explain your choice.

1.10 Task 5: Solution

Before a deciding, which model to use for the final evaluation, a rigourous comparision is carried out in terms of advantages and disadvantages of each model. Model 1 posits a direct proportional relationship between intended and observed resistance changes.

Advantages of Model 1:

- **Simplicity and Clarity:** With a single parameter (θ), Model 1 is not only easy to implement but also simplifies the interpretation of data, particularly beneficial in scenarios with straightforward, proportional changes.
- **Reduced Overfitting Risk:** The model's simplicity inherently guards against overfitting, making it robust when data is limited or particularly clean.

Disadvantages of Model 1:

- **Lack of Flexibility:** The absence of an intercept restricts the model's ability to adjust for any inherent bias in baseline resistance, potentially skewing results in scenarios with underlying offsets.
- **Limited Application:** Its simplicity, while advantageous in clean datasets, may not capture complexities in datasets where external factors influence baseline resistance.

On the other hand Model 2 enhances Model 1 by including an intercept, broadening the model's applicability.

Advantages of Model 2:

- **Greater Flexibility:** The addition of θ_0 allows the model to accommodate baseline shifts, thereby addressing systematic biases more effectively.
- **Comprehensive Fit:** The intercept enables Model 2 to capture a broader array of behaviors, enhancing the accuracy of the fit across varied scenarios.

Disadvantages of Model 2:

- **Increased Complexity:** The introduction of an additional parameter necessitates more extensive data for reliable parameter estimation and increases the risk of fitting noise.
- **Higher Computational Demand:** The computational requirements for Model 2 are greater, which may not be ideal in time-sensitive or resource-constrained environments.

Despite the increased complexity and the higher computational demand, model 2 was taken since the data points in the figures 2 and 2 indicate that there is indeed a baselineshift present. So for this it is important to consider each memristor's data step-by-step, highlighting how Model 2's parameters (θ_0 and θ_1) provide a more nuanced understanding of each memristor's behavior compared to Model 1's single parameter (θ).

- Memristor 1 **Model 1:** $\theta = -1.68$ suggests that the resistance change is discordant, decreasing instead of increasing as intended.
  Model 2: $\theta_0 = -74.85$ and $\theta_1 = -1.55$ indicate not only a consistent discordant behavior (similar to Model 1) but also a significant negative offset. This negative offset suggests an inherent lower baseline resistance, which Model 1 fails to capture. This additional detail from Model 2 can be crucial for applications requiring precise initial condition settings.
- Memristor 2 **Model 1:** $\theta = 0.63$ implies the resistance changes are less than intended but in the correct direction.
  Model 2: $\theta_0 = -103.78$ and $\theta_1 = 0.86$ show a considerable negative initial resistance and a slightly better responsiveness than suggested by Model 1. The significant negative intercept from Model 2 reveals an underestimation of the baseline resistance that Model 1 entirely overlooks, providing a more complete understanding of its operational characteristics.

- Memristor 3 **Model 1:** $\theta = -0.84$ indicates resistance decreases when it should increase, suggesting a discordant fault.
  Model 2: $\theta_0 = 43.55$ and $\theta_1 = -0.90$ reveal that the memristor starts with a higher baseline resistance and consistently exhibits discordant behavior. Model 2 not only confirms the discordant nature observed in Model 1 but also indicates a higher starting point, which is critical for settings where the resting state of resistance is pivotal.

- Memristor 4 **Model 1:** $\theta = 1.84$ shows an over-responsive behavior to changes.
  Model 2: $\theta_0 = -168.36$ and $\theta_1 = 2.19$ suggest an extremely low baseline resistance but with a very high responsiveness. This memristor's behavior in Model 2 highlights a severe initial deficit in resistance that enhances significantly upon activation, a detail Model 1 cannot show.

- Memristor 5 **Model 1:** $\theta = -0.05$ almost suggests a stuck behavior with very little resistance change.
  Model 2: $\theta_0 = -70.47$ and $\theta_1 = 0.04$ indicate a low baseline resistance with minimal change. Here, Model 2 provides essential insights into the near-stuck behavior while accounting for the negative baseline resistance, enhancing fault diagnosis beyond what Model 1 can achieve.

- Memristor 6 **Model 1:** $\theta = -0.65$ suggests a resistance decrease rather than an increase.
  Model 2: $\theta_0 = -172.33$ and $\theta_1 = -0.38$ confirm a strong negative baseline and discordant response, providing additional details on the extent of negative offset and the direction of change, which are overlooked in Model 1.

- Memristor 7 **Model 1:** $\theta = 0.01$ indicates almost no change in resistance.
  Model 2: $\theta_0 = 26.08$ and $\theta_1 = -0.03$ offer insights into a positive baseline shift with a slight decrease in resistance. This detailed observation from Model 2 is vital for accurately setting control systems that rely on precise baseline resistance values.

- Memristor 8 **Model 1:** $\theta = 1.14$ indicates a slightly over-responsive behavior.
  Model 2: $\theta_0 = -41.50$ and $\theta_1 = 1.23$ suggest a low initial resistance but a higher than expected change rate. Model 2's parameters provide a clearer picture of how the memristor responds to stimuli, including initial deficits that could impact operational efficiency.

The memristor's behavior, as detailed above, shows that Model 2 not only captures the proportional changes in resistance observed in Model 1 but also accounts for significant initial conditions (baseline shifts) that Model 1 entirely misses. This comprehensive capability makes Model 2 far superior for applications requiring accurate and detailed memristor behavior modeling.

1.11 Task 6: Problem

Finally, implement in the code (using if-statements and thresholds of your choice) how the parameters (of either Model 1 or Model 2, depending on your choice in the previous step) can be used to decide on the memristor fault type. In the report, describe how you did it (or include a screenshot of your code). Report (provide a list with) the memristor id and the fault type assigned to it according to your code output.

1.12 Task 6: Solution

Here a detailed analysis of the function `classify_memristor_fault_with_model2` is provided, which uses parameters from Model 2 to determine the type of faults in memristors based on their operational characteristics. The function `classify_memristor_fault_with_model2` classifies memristor faults by analyzing the slope parameter (θ_1) from a linear regression model. This approach focuses on identifying how closely the actual behavior of a memristor matches its expected behavior.

The function employs several conditional statements with specific thresholds to categorize faults:

- **Ideal Fault Detection:**

$$\text{if } \theta_1 > 0 \text{ and } |\theta_1 - 1.0| < 0.3$$

This condition checks if θ_1 is within 30% of the ideal value, which indicates nearly perfect behavior. This threshold tolerates minor variations that might arise due to imperfections in the memristor or external influences, assuming these do not significantly impact the device's functionality.

- **Discordant Fault Identification:**

$$\text{if } \theta_1 < 0 \text{ and } |\theta_1| > 0.3$$

This checks for negative θ_1 values that significantly deviate from zero, signifying that the resistance changes in the opposite direction to what is intended. The threshold of 0.3 ensures that only substantial negative deviations are considered faults, thus avoiding misclassification of minor anomalies as serious faults.

- **Concordant Fault Classification:**

$$\text{if } (0.3 \leq \theta_1 \leq 0.7) \text{ or } (\theta_1 > 1.3)$$

This range identifies cases where the memristor underperforms or overperforms relative to the desired changes. The lower limit of 0.3 and the upper limit of 1.3 provide a buffer to distinguish significant deviations that could affect performance, while excluding slight variations that do not materially alter device operation.

- **Stuck Fault Handling:**

$$\text{else}$$

This condition is the fallback for cases where θ_1 does not meet any of the specified conditions, typically implying a stuck behavior where the memristor shows negligible or no response to inputs.

While the function effectively categorizes faults, it could be enhanced by incorporating the intercept parameter (θ_0), which might provide insights into systematic biases or additional fault types, which are so far not covered in this assignment. Further, the exact values of the thresholds should ideally be determined based on empirical data specific to the memristor technology being used, to ensure that they accurately reflect the operational tolerances and specific requirements of the application. Furthermore with the values a deviation tolerance of the slope of 0.3 is incorporate which might be a bit too high for industry standards, but will demonstrate all fault cases here. The following table provides a summary of each memristor, their estimated parameters from a Model 2 regression analysis, and the fault type classification based on those parameters.

Table 1: Memristor Parameters and Fault Type Classification

Memristor ID	θ_0	θ_1	**Fault Type Assigned**
1	-74.85	-1.55	Discordant
2	-103.78	0.86	Ideal
3	43.55	-0.90	Discordant
4	-168.36	2.19	Concordant
5	-70.47	0.04	Stuck
6	-172.33	-0.38	Discordant
7	26.08	-0.03	Stuck
8	-41.50	1.23	Ideal

1.13 Bonus Task: Problem

Reconsider the task of computing θ_0^*, θ_1^* in Model 2. Instead of computing the individual partial derivatives, we will reformulate the objective in matrix-vector notation. Let $\boldsymbol{\theta} = (\theta_0, \theta_1)^T$, $\mathbf{x} = (\Delta R_1^{\text{ideal}}, \ldots, \Delta R_m^{\text{ideal}})^T \in$

$\mathbb{R}^m$ and $\mathbf{y} = (\Delta R_1, \ldots, \Delta R_m)^T \in \mathbb{R}^m$. We will define a matrix $\mathbf{X}$ such that we can re-write the minimization objective as

$$\boldsymbol{\theta}^* = \underset{\boldsymbol{\theta}}{\operatorname{argmin}} E(\boldsymbol{\theta}) \qquad \text{with} \quad E(\boldsymbol{\theta}) = 1/m \|\mathbf{X}\boldsymbol{\theta} - \mathbf{y}\|_2^2$$

Write down the definition of the design matrix $\mathbf{X}$ such that this objective is equivalent to Equation 2 and state the dimensions of $\mathbf{X}$. Using matrix calculus, derive an analytical expression for the solution $\boldsymbol{\theta}^*$ by computing the gradient of the error w.r.t. $\boldsymbol{\theta}$ and setting it to the zero vector, i.e., $\nabla_{\boldsymbol{\theta}} E(\boldsymbol{\theta}^*) = \mathbf{0}$. Your final expression for $\boldsymbol{\theta}^*$ should involve the Moore-Penrose pseudoinverse. Implement the computation for $\boldsymbol{\theta}^*$ in the function `bonus_fit_lin_model_with_intercept_using_pinv` by making use of numpy's `pinv` (read: pseudoinverse) function. You can test the functionality of your implementation by calling `bonus_fit_lin_model_with_intercept_using_pinv` instead of calling `fit_lin_model_with_intercept` in `main.py`. The resulting $\boldsymbol{\theta}$ should be equivalent. Clearly show all steps of your derivation in your report.

1.14 Bonus Task: Solution

In a linear model with an intercept, the design matrix X is set up to include:

- The first column, consisting of ones, represents the intercept term θ_0.
- The second column contains the predictor variables $\Delta R_{\text{ideal}_i}$ for each observation i.

$$X = \begin{bmatrix} 1 & \Delta R_{\text{ideal}_1} \\ 1 & \Delta R_{\text{ideal}_2} \\ \vdots & \vdots \\ 1 & \Delta R_{\text{ideal}_m} \end{bmatrix}$$

The matrix X has dimensions $m \times 2$, where m is the number of observations.

The objective function is expressed in matrix notation to minimize the squared differences between the observed values and predictions:

$$E(\theta) = 1/m * \|X\theta - y\|_2^2$$

Where:

- $X\theta$ produces the vector of predictions.
- y is the vector of observed values.
- $\|z\|_2^2$ is the squared Euclidean norm of vector z, calculated as $z^T z$.

The gradient of $E(\theta)$ with respect to θ is computed as:

$$\nabla_\theta E(\theta) = 2/m X^T (X\theta - y)$$

Setting the gradient to zero to find the optimal $\boldsymbol{\theta}^*$:

$$2/m X^T (X\theta^* - y) = 0$$

$$X^T X\theta^* = X^T y$$

The equation is rearranged to isolate θ^*:

$$\theta^* = (X^T X)^{-1} X^T y$$

assuming $X^T X$ is invertible. If $X^T X$ is not invertible, the Moore-Penrose pseudoinverse is used:

$$\theta^* = X^+ y$$

Where X^+ is defined as:

$$X^+ = \lim_{\delta \to 0} ((X^T X + \delta I)^{-1} X^T)$$

δ is a small positive value and I is the identity matrix of appropriate size.

This approach minimizes the residual sum of squares between the observed and predicted values, denoting the least squares estimates for the coefficients θ_0 and θ_1 as $\boldsymbol{\theta}^*$. In the implementation of the function `bonus_fit_lin_model_with_intercept_using_pinv`, a linear model with an intercept is fitted utilizing the Moore-Penrose pseudoinverse. This approach is especially beneficial in cases where the design matrix may not be of full rank, thereby precluding the direct inversion of $X^T X$.

The function is designed to accept two numpy arrays as parameters: `x`, representing the independent variable (ΔR_{ideal}), and `y`, the dependent variable (ΔR). These inputs are used to construct the design matrix X, which incorporates a column of ones to accommodate the intercept along with the predictor variables.

The Moore-Penrose pseudoinverse of the design matrix X is calculated using NumPy's `pinv` method within the function. This pseudoinverse is then employed to solve the linear equation $X\theta = y$ for θ, where θ consists of the coefficients θ_0 (intercept) and θ_1 (slope). The calculation of θ is performed through matrix multiplication between the pseudoinverse of X and the vector y.

Finally, the coefficients θ_0 and θ_1 are extracted from the resultant vector θ and returned as a tuple. This tuple represents the fitted parameters of the linear model, indicating the intercept and the slope of the best-fit line under the least squares criterion.

The function encapsulates a robust method for fitting a linear regression model, ensuring computational stability and precision even when traditional methods based on direct matrix inversion may fail. This attribute renders it particularly apt for practical applications involving real-world data that may exhibit issues such as multicollinearity or where the number of data points is insufficient relative to the number of predictors. In a comparison with the original implementation in the plot it delivers coherent the same results.

2 Logistic Regression

In the task we are provided with different datasets, their corresponding range values and functions depicting classes 0 and 1, which are to be predicted by our classifier. When creating the design matrices, the goal was to approximately capture the true formula of each function and add the calculated values as new features to the matrix. This will guide the classifier to more effectively identify patterns within the data, consequently improving accuracy.

For the design matrix of the first dataset, we have used the X_1 and X_2 values and an additional binary feature column. The two visible classes 0 and 1 are very easily differentiable by specifying the ranges of X_1 and X_2. More specifically, the binary feature will have a value of 1 if $X_1 \in [1, 29]$ and $X_2 \in [0, 20]$. Otherwise, the feature value will be zero.

The second design matrix resembles a circle with the center in [0,0]. The analytical formula of such a circle is $X_1^2 + X_2^2 = r^2$, with r being the radius. Therefore, we will add $X_1^2 + X_2^2 - 24^2$ as the value of the additional feature column.

Lastly, the third figure models a polynomial function. We have represented it with $\cos(2.5X_1) + \frac{X_1^3}{4} + \frac{X_1^2}{3}$ and added 3 additional feature columns for the cosine, square and cubic operation.

The data is then split for training and testing, after which we create the classifier, fit the model to the data and calculate the cross-entropy loss. The accuracy and cross-entropy loss of the models is provided below.

- Dataset 1: Train accuracy: 100%, Test accuracy: 100%,
- Dataset 2: Train accuracy: 100%, Test accuracy: 100%,
- Dataset 3: Train accuracy: 93.49%, Test accuracy: 97.20%.

- Dataset 1: Train loss: 0.02437368, Test loss: 0.02256633,

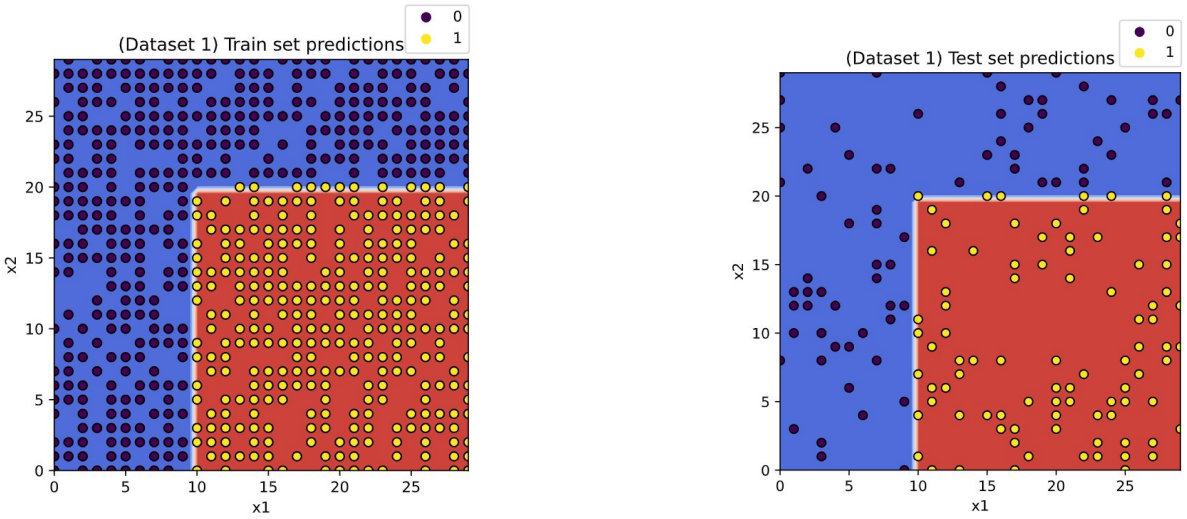
- Dataset 2: Train loss: 0.00105729, Test loss: 0.00182642,
- Dataset 3: Train loss: 0.22199969, Test loss: 0.21136677.

The final parameters and the bias terms of our models are as follows:

- Dataset 1: Weights: [0.14187233 -0.18127346 6.2323964], Bias: -2.58906992,
- Dataset 2: Weights [0.35833391 0.35951744 -0.48184566], Bias: 0.02400595,
- Dataset 3: Weights: [-1.79294649 -6.88425721 1.28053444 -0.94210303 2.81971318 -1.02370825], Bias: -4.36099138.

When creating the Logistic Regression classifier, we have used the l2 penalty and `liblinear` solver. Other combinations of solvers and penalties decrease the accuracy level of dataset 2 or dataset 3 test entries. For example, the l1 penalty gives 99.38% accuracy for the dataset 2 test data. The `elasticnet` penalty provides 100% accuracy in that case, however with higher loss values. Furthermore, the l2 penalty with `liblinear` was the only combination that increased the test accuracy of the third dataset to 97.20%. Others provided only 96.50%. The l1 regularization is most beneficial in larger datasets with many features, because of its tendency to drive some of the model coefficients to zero. This can simplify the model and sometimes improve accuracy. However, in our data, all coefficients are important and therefore the l1 penalty would not contribute much. The `elasticnet` penalty, which includes both l1 and l2 was a slightly better match, but not as good as l2 alone. The l2 regularization shrinks the coefficients but does not set them to zero. This helps in reducing model complexity and preventing overfitting, particularly when the dataset has highly correlated features, which is present in our data.

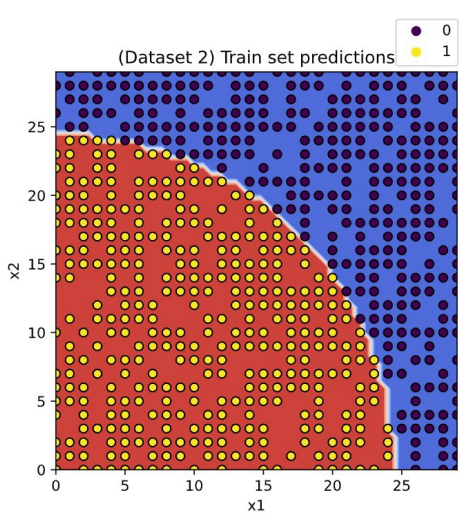
The generated train and test plots with `plot_logistic_regression`:



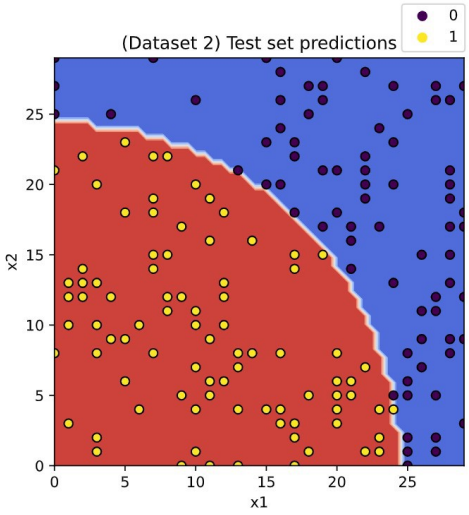
(a) Training data of Dataset 1

(b) Testing data of Dataset 1

Figure 4: Decision boundary plots of the logistic regression model

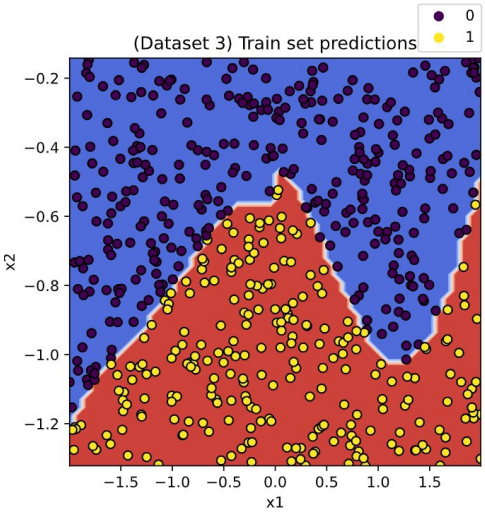


(a) Training data of Dataset 2

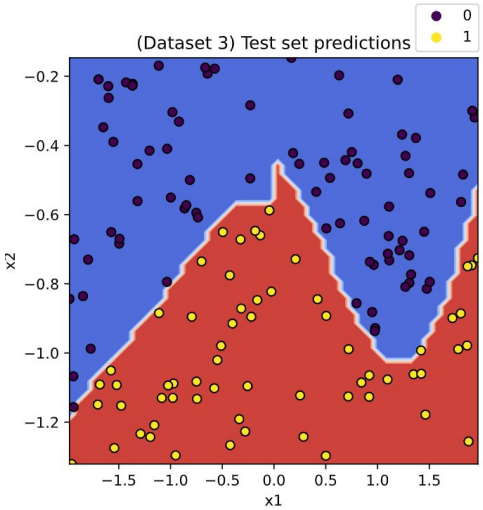


(b) Testing data of Dataset 2

Figure 5: Decision boundary plots of the logistic regression model



(a) Training data of Dataset 3



(b) Testing data of Dataset 3

Figure 6: Decision boundary plots of the logistic regression model

At the end we are given the following question:

Assume we have trained a logistic regression classifier (binary classes) and are given a test dataset $\mathcal{D}$. Is the following statement correct? "If the classifier predicts the correct class for *all* elements in $\mathcal{D}$ (100% accuracy), then it follows that the cross-entropy loss (w.r.t. $\mathcal{D}$) is 0." Explain your reasoning.

Answer

The statement is not correct. Accuracy and loss are two different metrics for measuring the quality of our model and values of one do not directly effect the values of the other. Therefore, 100% accuracy on the test

dataset does not guarantee that the cross-entropy loss will be zero.

Accuracy is simply the count of misclassifications made on the data. The larger the accuracy, the fewer mistakes are made. On the other hand, cross-entropy loss is a measure of how well the predicted probabilities match the actual labels in a classification task. The larger the loss, the larger the errors are made on the data.

Getting 100% accuracy just means that all predicted labels match the true labels. However, it does not guarantee that the predicted probabilities for each class are perfectly calibrated. For such an accuracy we could still get a very high loss value, which would mean that the model made large errors on a few data items. The most optimal case is when we get a small loss and large accuracy. This means that the model made small and very few or no errors on the data.

For example, let's say a model has accuracy of 0% and the true y labels are [0, 0, 1, 1] and the predicted probabilities are [0.9, 0.9, 0.1, 0.1]. It is clear that the model did not correctly predict any of the labels because pf the inaccurate probabilities. The model was 90% certain that the first item will belong to the specific class, which of course is not the case. On the other hand it only 10% sure that the item will belong to the third class. A example like this will have a higher loss value because the probabilities are far from correct.

However, if we modify the probabilities and use [0.6, 0.6, 0.4, 0.4] as an example, then the loss reduces. We can notice the the accuracy is still the same and 0%. However, the probabilities are significantly closer to reaching the correct classification, compared to the first example. The second example will have a lower loss even though the accuracy stayed the same.

This indicates that the accuracy does not determine the loss value and possibly having 100% accuracy, does not necessarily get zero as the loss. Loss is determined by the probabilities, which we do not know just from looking at the accuracy.

3 Gradient descent

Consider the following function $f : \mathbb{R}^2 \to \mathbb{R}$ (called the *Ackley* function):

$$f(x, y) = -20e^{-0.2\sqrt{0.5(x^2+y^2)}} - e^{0.5(cos(2\pi x)+cos(2\pi y))} + e + 20. \tag{3}$$

We want to minimize this function using the *Gradient Descent* algorithm, i.e., we wish to find

$$(x^*, y^*) = \underset{x,y}{\operatorname{argmin}}\, f(x, y).$$

The global minimum of this function is at the point $(0, 0)$ and you should be able to find it by running the Gradient Descent algorithm.

3.1 Task 1:Problem

In the code, implement the gradient descent algorithm (function `gradient_descent`).

3.2 Task 1:Solution

The provided code implements the gradient descent algorithm to optimize a function $f(x, y)$ in $\mathbb{R}^2$. This method iteratively adjusts the parameters x and y, targeting the local minimum of the function by navigating in the direction opposite to its gradient. Essential parameters for this function include the target function f, its gradient df, initial coordinates x_0 and y_0, a learning rate α, a decay factor for the learning rate λ, and the maximum number of iterations N. The selection of the starting points and the learning rate

critically influences the algorithm's efficiency, particularly in complex optimization landscapes that may feature multiple local minima.

The algorithm initiates with x and y at x_0 and y_0. In each iteration, the gradient at these coordinates is computed. Updates to x and y are performed by moving against the gradient, scaled by the current learning rate α. This rate is subsequently reduced by multiplying it with λ after each iteration, thus enhancing the precision of the approach as iterations progress. The function values at each point are stored in an array f_{list} to facilitate post-hoc analysis of the convergence behavior.

Upon algorithm completion, the function outputs the coordinates of the presumed minimum alongside a detailed log of function values across iterations. This output is crucial for assessing the descent's efficiency and understanding the pattern of convergence. Ensuring accuracy in the gradient calculations is vital, as errors can significantly hinder convergence. Further refinements, such as fine-tuning the learning rate or implementing additional convergence criteria like thresholds for minimal changes in function values, could optimize computational performance and robustness.

3.3 Task 2:Problem

In the code, implement the Ackley function (function `ackley`).

3.4 Task 2:Solution

The provided Python implementation computes the Ackley function, a renowned benchmark function in the field of optimization, evaluated at any given point (x, y) in $\mathbb{R}^2$. The Ackley function is extensively utilized to assess the performance of optimization algorithms, due to its challenging landscape that features a significant number of local minima.

The function, named `ackley`, is formulated to compute values based on the equation:

$$f(x,y) = -20\exp\left(-0.2\sqrt{0.5\left(x^2+y^2\right)}\right) - \exp\left(0.5\left(\cos(2\pi x)+\cos(2\pi y)\right)\right) + e + 20.$$

This calculation involves two main components: an exponential decay that is influenced by the Euclidean distance from the origin, which contributes to the function's multimodal nature, and a cosine component that introduces oscillations, adding further complexity to the optimization landscape.

The function parameters, x and y, serve as inputs representing the coordinates in $\mathbb{R}^2$. The first computational part, termed `part1`, addresses the exponential decay shaped significantly by the distance from the origin. The second part, `part2`, incorporates periodic complexity through cosine terms, essential for testing the algorithm's ability to navigate and escape local optima. The final output, `value`, merges these calculated components with constants to properly scale and center the function.

This implementation of the Ackley function is crucial for benchmarking the capabilities of optimization algorithms, offering a complex topology with multiple local minima to navigate. The use of efficient NumPy functions ensures both high performance and numerical precision, rendering the Ackley function a valuable tool in optimization research. It aids in evaluating the robustness and effectiveness of various algorithms in locating global minima within complex landscapes. A plot of this function can be seen in Figure 7.

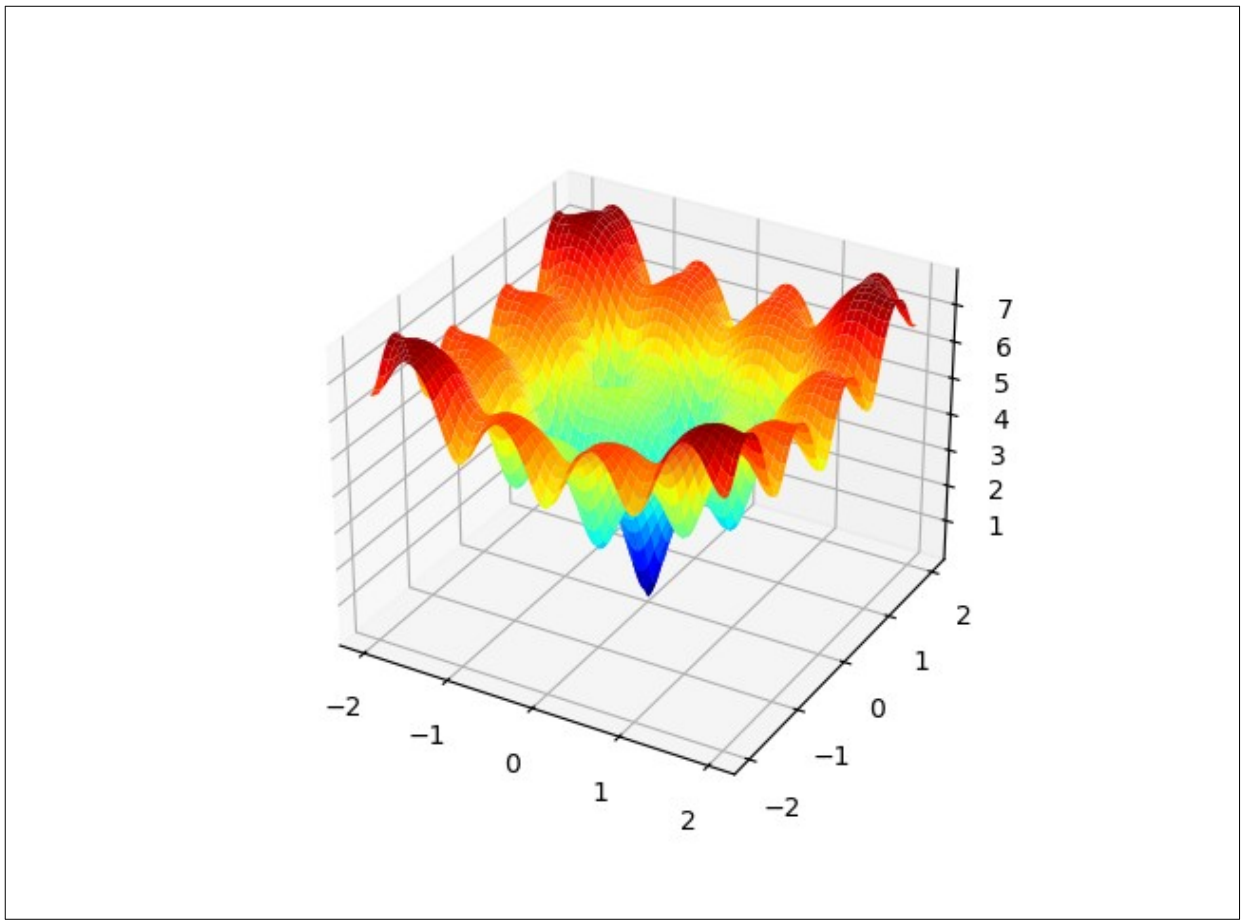


Figure 7: 3D Plot of Ackley function

3.5 Task 3: Problem

Using pen and paper, calculate the partial derivatives of f (with respect to x and y), i.e., $\frac{\partial f}{\partial x}$ and $\frac{\partial f}{\partial y}$. Implement the expressions you have derived in the code (function `gradient_ackley`). Include all steps of your derivation in the report.

3.6 Task 3: Solution

The Ackley function, frequently used as a test function in optimization problems, is defined as:

$$f(x, y) = -20 \exp\left(-0.2\sqrt{0.5(x^2 + y^2)}\right) - \exp\left(0.5(\cos(2\pi x) + \cos(2\pi y))\right) + e + 20.$$

To derive the partial derivative with respect to y, we apply differential calculus principles, focusing on the chain rule, to decompose the function into simpler components. First, consider the component:

$$g(x, y) = -20 \exp\left(-0.2\sqrt{0.5(x^2 + y^2)}\right).$$

Introduce intermediate variables for simplification:

$$u(x, y) = 0.5(x^2 + y^2), \quad v(u) = \sqrt{u}, \quad w(v) = \exp(-0.2v).$$

The derivative of $g(x, y)$ with respect to y is found by chaining the derivatives:

$$\frac{\partial g}{\partial y} = \frac{\partial g}{\partial w}\frac{\partial w}{\partial v}\frac{\partial v}{\partial u}\frac{\partial u}{\partial y}$$

where:

$$\frac{\partial u}{\partial y} = y, \quad \frac{\partial v}{\partial u} = \frac{1}{2\sqrt{u}} = \frac{1}{2\sqrt{0.5(x^2+y^2)}}, \quad \frac{\partial w}{\partial v} = -0.2\exp(-0.2\sqrt{0.5(x^2+y^2)}), \quad \frac{\partial g}{\partial w} = -20.$$

Thus, this derivative simplifies to:

$$\frac{\partial g}{\partial y} = 2\exp\left(-0.2\sqrt{0.5(x^2+y^2)}\right)\frac{y}{\sqrt{0.5(x^2+y^2)}}.$$

The second component of the function is:

$$h(x,y) = -\exp\left(0.5(\cos(2\pi x)+\cos(2\pi y))\right).$$

Defining:

$$p(x,y) = \cos(2\pi x)+\cos(2\pi y),$$

and applying the chain rule:

$$\frac{\partial h}{\partial y} = \frac{\partial h}{\partial p}\frac{\partial p}{\partial y}$$

where:

$$\frac{\partial p}{\partial y} = -2\pi\sin(2\pi y), \quad \frac{\partial h}{\partial p} = -0.5\exp(0.5p(x,y)).$$

Hence, this derivative evaluates to:

$$\frac{\partial h}{\partial y} = \pi\exp\left(0.5(\cos(2\pi x)+\cos(2\pi y))\right)\sin(2\pi y).$$

Combining the derivatives of both components, the total partial derivative of the Ackley function with respect to y is:

$$\frac{\partial f}{\partial y} = 2\exp\left(-0.2\sqrt{0.5(x^2+y^2)}\right)\frac{y}{\sqrt{0.5(x^2+y^2)}} + \pi\exp\left(0.5(\cos(2\pi x)+\cos(2\pi y))\right)\sin(2\pi y).$$

For the derivation with respect to x the approach is analogue as before, let's decompose the function into manageable parts for differentiation. The first part:

$$g(x,y) = -20\exp\left(-0.2\sqrt{0.5(x^2+y^2)}\right)$$

involves variables:

$$u(x,y) = 0.5(x^2+y^2), \quad v(u) = \sqrt{u}, \quad w(v) = \exp(-0.2v).$$

Applying the chain rule to compute $\frac{\partial g}{\partial x}$:

$$\frac{\partial g}{\partial x} = \frac{\partial g}{\partial w}\frac{\partial w}{\partial v}\frac{\partial v}{\partial u}\frac{\partial u}{\partial x},$$

where the derivatives of the intermediate variables are:

$$\frac{\partial u}{\partial x} = x, \quad \frac{\partial v}{\partial u} = \frac{1}{2\sqrt{u}} = \frac{1}{2\sqrt{0.5(x^2+y^2)}}, \quad \frac{\partial w}{\partial v} = -0.2\exp(-0.2\sqrt{0.5(x^2+y^2)}), \quad \frac{\partial g}{\partial w} = -20.$$

Thus, the detailed derivative simplifies to:

$$\frac{\partial g}{\partial x} = 2\exp\left(-0.2\sqrt{0.5(x^2+y^2)}\right)\frac{x}{\sqrt{0.5(x^2+y^2)}}.$$

Turning to the second component:

$$h(x,y) = -\exp\left(0.5(\cos(2\pi x)+\cos(2\pi y))\right)$$

and defining:

$$p(x, y) = \cos(2\pi x) + \cos(2\pi y),$$

the chain rule gives:

$$\frac{\partial h}{\partial x} = \frac{\partial h}{\partial p}\frac{\partial p}{\partial x}$$

with:

$$\frac{\partial p}{\partial x} = -2\pi \sin(2\pi x), \quad \frac{\partial h}{\partial p} = -0.5 \exp(0.5p(x, y)).$$

This results in:

$$\frac{\partial h}{\partial x} = \pi \exp\left(0.5(\cos(2\pi x) + \cos(2\pi y))\right) \sin(2\pi x).$$

Combining these, the full expression for the partial derivative of the Ackley function with respect to x is:

$$\frac{\partial f}{\partial x} = 2 \exp\left(-0.2\sqrt{0.5(x^2 + y^2)}\right) \frac{x}{\sqrt{0.5(x^2 + y^2)}} + \pi \exp\left(0.5(\cos(2\pi x) + \cos(2\pi y))\right) \sin(2\pi x).$$

This mathematical exploration emphasizes optimization strategies such as gradient descent, where these derivatives provide the directionality needed for effectively minimizing the function, guiding convergence towards the function's global minimum, ideally at $(0, 0)$.

In the *gradient_ackley* function in the code, the gradient of the Ackley function is calculated at any point (x, y). This function is essential for optimization algorithms, utilizing gradient data to navigate the function's landscape. The code computes *exp_component* for the exponential decay based on the Euclidean distance from the origin, reflecting rapid changes near the minimum, and *cos_component* for the exponential of cosine terms, which introduces local minima. Partial derivatives *df_dx* and *df_dy* are determined for the x and y coordinates, respectively, incorporating both exponential and oscillatory components of the function. These derivatives form a gradient vector, crucial for guiding optimization algorithms towards the function's minimum. The implementation leverages numpy for efficient and accurate numerical computations, critical in performance-sensitive optimization tasks.

3.7 Task 4:Problem

Choose initial points x_0, y_0 by sampling them from a standard normal distribution, i.e.,

$$x_0 \sim \mathcal{N}(0, 1), \; y_0 \sim \mathcal{N}(0, 1)$$

where $\mathcal{N}(0, 1)$ denotes a normal distribution with zero-mean ($\mu = 0$) and unit variance ($\sigma^2 = 1$). To make your code reproducable, we set numpy's random seed to a constant (which will result in the same samples x_0, y_0 in every run of your script).

3.8 Task 4:Solution

To initialize the starting points x_0 and y_0 for the gradient descent algorithm, the script samples values from a standard normal distribution ($\mu = 0$, $\sigma^2 = 1$) using NumPy's random functions. To ensure reproducibility of results across multiple runs, the random seed is set to a constant value, but later in the code. This setup guarantees that the initial conditions are consistent, which is crucial for comparing the performance of the algorithm under controlled conditions of randomness. The sampled values are formatted to four decimal places, providing precise and clear numerical output. This method not only facilitates debugging but also enables a systematic evaluation of how starting positions influence the optimization process.

1. Choose hyperparameters of the gradient descent algorithm (i.e., number of iterations, learning rate, and learning rate decay) and minimize the Ackley function. Report the parameters that you have used.

2. Generate a plot showing how the cost changes over iteration. Include it in the report.

3.9 Task 5: Problem

Choose hyperparameters of the gradient descent algorithm (i.e., number of iterations, learning rate, and learning rate decay) and minimize the Ackley function. Report the parameters that you have used.

3.10 Task 5:Solution

Minimizing the Ackley function through the gradient descent algorithm requires an astute selection of hyperparameters due to the function's intricate landscape characterized by numerous local minima. This report details the rationale and effects of the chosen hyperparameters: the learning rate, learning rate decay, and the number of iterations.

The learning rate, set at 0.4, is a crucial hyperparameter in the gradient descent algorithm. It directly influences the magnitude of steps taken towards the minimum. The Ackley function, known for its pronounced valleys and extensive flat regions, necessitates a balance in step size. A moderately high learning rate ensures rapid escape from local minima and adequate exploration of the search space, while still being conservative enough to avoid overshooting the global minimum. This balance is critical for ensuring both the progress and stability of the search process.

Learning rate decay is implemented at a factor of 0.5, which systematically reduces the learning rate throughout the iterations. This adaptive adjustment is vital as it allows for larger initial steps, beneficial for exploring the global structure of the Ackley function, and smaller, more precise steps as the minimum is approached. Such fine-tuning is especially important in the later stages of the search, preventing the oscillation around the minimum—a common challenge when the landscape near the minimum is relatively flat.

The choice of fifteen iterations strikes a balance between computational efficiency and search depth. Given the initial learning rate and the decay mechanism, this number allows the algorithm sufficient iterations to potentially converge towards the minimum without incurring excessive computational costs. It is a strategic decision to ensure that each iteration, with progressively smaller steps due to the learning rate decay, effectively contributes to locating a more accurate minimum.

The implementation of these hyperparameters involves initializing the algorithm at a point (x_0, y_0), followed by iterative updates based on the gradient of the Ackley function and the dynamically adjusted learning rate. After the predefined number of iterations, the algorithm outputs the coordinates of the identified minimum, reflecting the effectiveness of the hyperparameter settings in navigating the challenging optimization landscape of the Ackley function.

Setting the learning rate to 1 for gradient descent optimization of the Ackley function resulted in converging to the point $(-3.4082, 3.4816)$ with a function value of 12.2768, indicating suboptimal convergence away from the global minimum. This high learning rate may lead to overshooting the minimum or diverging from optimal valleys, as the algorithm settled on a point significantly distant from the global minimum at $(0, 0)$.

3.11 Task 6: Problem

Generate a plot showing how the cost changes over iteration. Include it in the report.

3.12 Task 6: Solution

In the code it is utilized the Matplotlib library to generate a plot of the Ackley function values across iterations during a gradient descent optimization. The code initiates a new figure, plots the values stored in f_list (each corresponding to an iteration of the gradient descent), and labels the axes and the plot for clarity. This visual representation is crucial for analyzing the effectiveness of the optimization, highlighting trends such as decreases or plateaus in function values which indicate convergence or stagnation. Additionally, the code prints the function value at the final solution and compares it to the global optimum, offering immediate quantitative feedback on the algorithm's performance. This combination of graphical and textual output is essential for comprehensively assessing and reporting on the optimization strategy's outcomes in a compact and informative manner. In Figure 8 the cost function for 15 iterations, a learning rate decay of 0.5 and a learning rate of 0.4 is plotted and in Figure 9 the learning rate decay is set to 1.

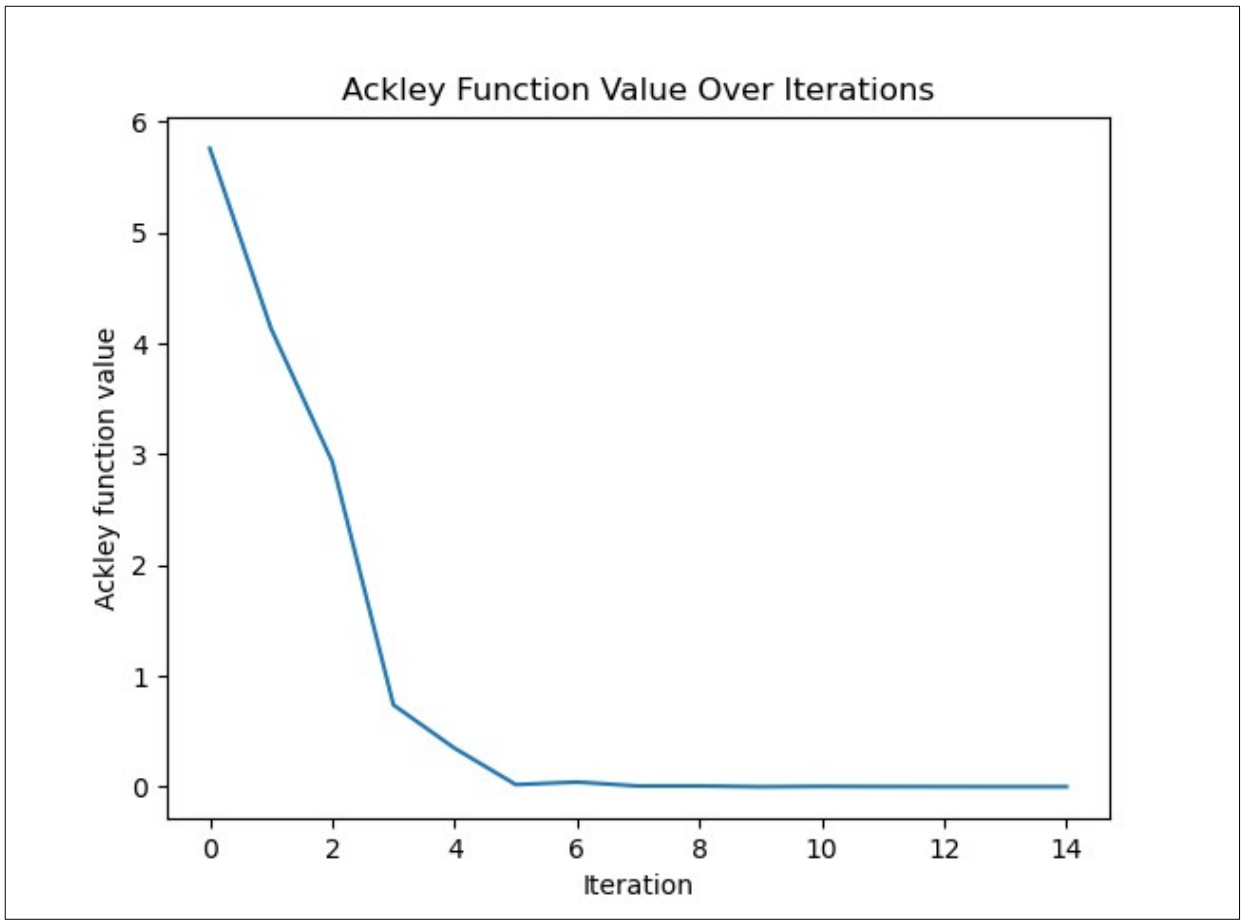


Figure 8: Ackley function value over iterations of the gradient decent algorithm for a learning rate of 0.4, a learning rate decay of 0.5 and 15 iterations.

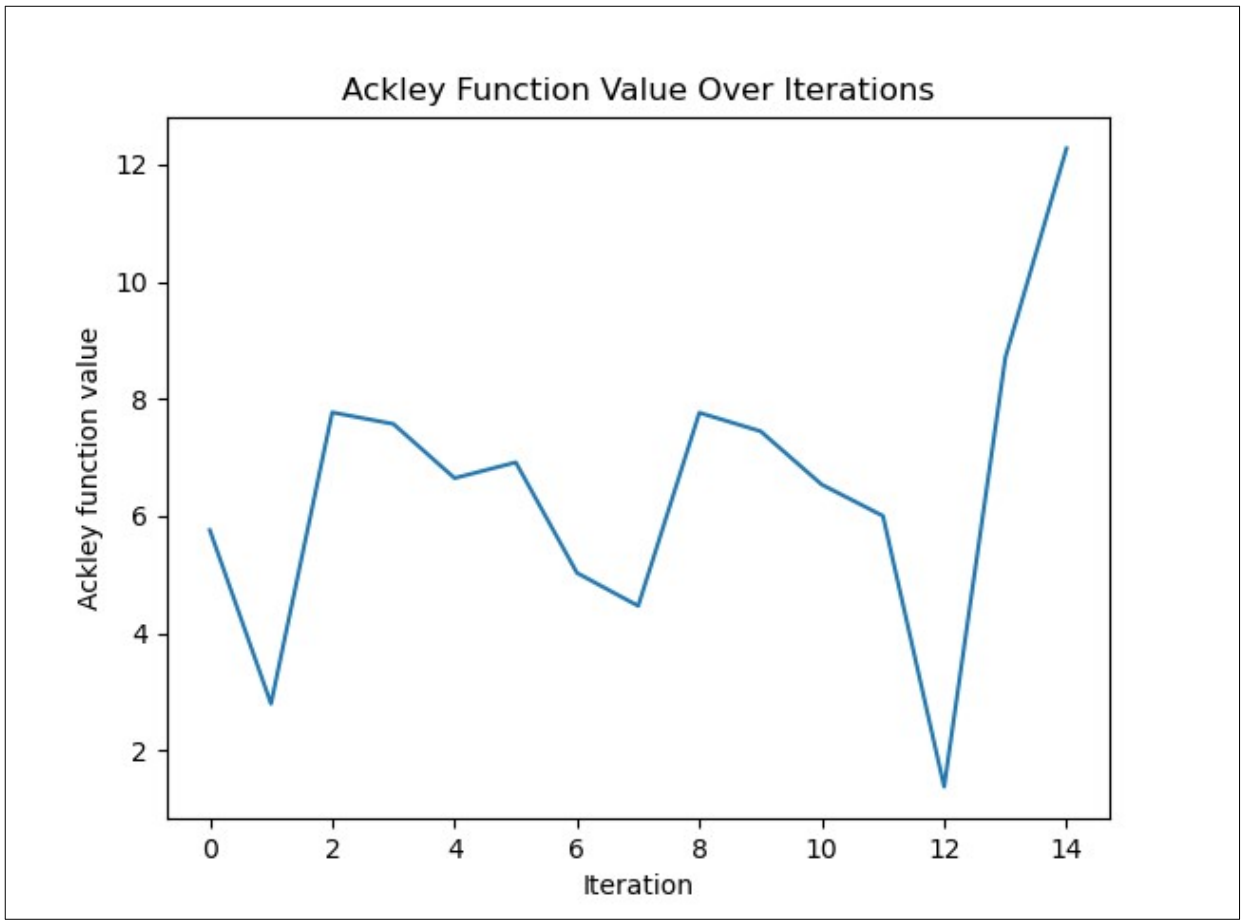


Figure 9: Ackley function value over iterations of the gradient decent algorithm for a learning rate of 0.4, a learning rate decay of 1 and 15 iterations.

The comparison between the two graphs underscores the impact of learning rate decay in gradient descent optimization. While a constant learning rate (decay = 1) may cause instability or divergence in complex landscapes like that of the Ackley function, introducing a decay factor (decay = 0.5) enhances convergence by adapting the step size according to the proximity to the minimum. The first approach struggles with stabilization, as seen in the fluctuating and ultimately rising function values, whereas the second approach, with its methodical reduction in step size, leads to a more stable and successful optimization trajectory.

3.13 Task 7:Problem

Why is this function challenging to optimize?

3.14 Task 7:Solution

The Ackley function serves as a rigorous benchmark in optimization, challenging the capabilities of optimization algorithms due to its intricate landscape. Below are the key reasons outlining why optimizing this function is notably difficult:

- **Abundance of Local Minima:** The periodic nature of the cosine components within the Ackley function results in numerous local minima. This trait frequently ensnares optimization algorithms, particularly those dependent on gradient information, preventing them from converging to the global minimum.

- **Extensive Plateau Regions:** The function's landscape features areas with negligible gradient magnitudes, particularly distant from the origin. This characteristic hampers gradient-based optimization techniques, which depend on substantial gradient magnitudes to navigate. Near-zero gradients lead to minimal positional updates, potentially causing the algorithm to become stagnant.
- **Intricate Composition of Exponential and Trigonometric Functions:** The Ackley function integrates an exponential term modulated by the square root of the sum of squares of the input variables, alongside trigonometric terms. This configuration causes abrupt gradient variations, making the journey towards the minimum erratic and challenging.
- **Sensitivity to Initial Positions:** The starting location significantly influences the efficiency and outcome of the optimization process. Some initial positions may straightforwardly lead to the global minimum, whereas others might end up in local minima or extensive flat areas, making progress slow and computationally intensive.
- **Complications Due to Increased Dimensionality:** As the number of dimensions increases, the landscape of the Ackley function becomes even more daunting, with an exponential increase in local minima. This not only adds to the computational load but also enhances the likelihood of encountering misleading local optima, amplifying the overall challenge of the optimization task.

Together, these elements confirm the Ackley function as a comprehensive evaluation tool for testing the efficacy of optimization algorithms across complex, multimodal landscapes.

3.15 Task 8:Problem

Briefly discuss what happens when you set the parameter `lr_decay` to 1.0 (i.e., the learning rate stays the same in all iterations). Why is it very helpful to have a learning rate that decays when optimizing the Ackley function?

3.16 Task 8:Solution

In gradient descent optimization, the learning rate (η) critically determines the step size towards the minimum of a function. Setting `lr_decay` $= 1.0$ keeps the learning rate constant, posing unique challenges, particularly for complex functions like the Ackley function, characterized by multiple local minima and large flat areas.

Challenges with a Constant Learning Rate:

- **Oscillation:** A high constant learning rate may lead to continuous overshooting of the minimum, resulting in oscillations around it without convergence. This is especially problematic on the steep slopes of the Ackley function, where gradients can abruptly change.
- **Slow Convergence in Flat Areas:** Conversely, a low constant learning rate may not overcome the minimal gradient values in flat areas, significantly slowing down the convergence or causing stagnation.

Benefits of a Decaying Learning Rate: A decaying learning rate (`lr_decay` < 1.0) dynamically adjusts the step size, enhancing the optimization efficiency in complex landscapes:

- **Dynamic Step Sizing:** The algorithm starts with a larger learning rate for rapid initial progress and reduces the rate to fine-tune the approach as it nears the minimum. This is crucial for escaping local minima and efficiently navigating flat regions.
- **Preventing Oscillations:** Reducing the step size as the algorithm approaches the minimum helps stabilize the optimization by avoiding overshooting, critical in achieving convergence on complex landscapes.
- **Enhanced Convergence in Flat Areas:** The controlled reduction in the learning rate ensures sufficient momentum is maintained to progress out of flat areas without premature stalling.

Algorithmic Detail of Learning Rate Decay: The learning rate is updated at each iteration by multiplying it with a decay factor less than 1 (`lr_decay`):

$$\eta_{t+1} = \eta_t \cdot \texttt{lr\_decay}$$

This exponential decay provides a controlled reduction in step size, adapting to the topology of the Ackley function and fine-tuning the updates based on the encountered landscape.

In summary, a decaying learning rate is essential for navigating the complex, multimodal landscapes typical of the Ackley function, enhancing both the efficacy and reliability of the convergence process.

3.17 Bonus task:Problem

Let $g : \mathbb{R} \to \mathbb{R}$ be a function whose derivative (w.r.t. its input) $\frac{dg}{dx}$ exists everywhere. To minimize g using the Gradient Descent algorithm, we must have access to the derivative of the function at an arbitrary point. Assume that we do not know the functional form of g, but we can only *evaluate* it at arbitrary points[1] (i.e., we can compute $g(x)$ for any $x \in \mathbb{R}$). How can we *approximate* $\frac{dg}{dx}\Big|_{x=x'}$ for an arbitrary point $x' \in \mathbb{R}$? Write down the definition of the derivative and explain how such a general approximation can be computed.

3.18 Bonus task: Solution

To implement the Gradient Descent algorithm for minimizing a function $g : \mathbb{R} \to \mathbb{R}$ without direct access to its derivative, an effective strategy involves using the central difference formula for approximating the derivative at any point x'. This method is grounded in Taylor series expansions and leverages the symmetry in derivatives.

Central Difference Formula: The derivative of g at $x = x'$ using the central difference method is approximated by:

$$\left.\frac{dg}{dx}\right|_{x=x'} \approx \frac{g(x'+h) - g(x'-h)}{2h}$$

where h is a small step size.

Taylor Series Expansion: Assume g is smooth enough (at least C^3). The Taylor series expansions around x' are given by:

$$g(x'+h) = g(x') + hg'(x') + \frac{h^2}{2}g''(x') + \frac{h^3}{6}g'''(\xi_1)$$

$$g(x'-h) = g(x') - hg'(x') + \frac{h^2}{2}g''(x') - \frac{h^3}{6}g'''(\xi_2)$$

where $\xi_1, \xi_2 \in (x'-h, x'+h)$.

Derivative Approximation: By subtracting these expansions and simplifying, we obtain:

$$g(x'+h) - g(x'-h) = 2hg'(x') + \frac{h^3}{6}(g'''(\xi_1) - g'''(\xi_2))$$

Dividing through by $2h$, the approximation becomes:

$$\frac{g(x'+h) - g(x'-h)}{2h} = g'(x') + \frac{h^2}{12}(g'''(\xi_1) - g'''(\xi_2))$$

This shows the error term is $O(h^2)$, confirming the second-order accuracy.

Error Analysis and Optimal h: Optimal h balancing truncation and round-off errors is:

$$h = \left(\frac{3\epsilon|g'(x')|}{|g'''(x')|}\right)^{1/3}$$

where ϵ denotes the machine precision.

[1] We effectively treat g as a black box, which makes this method very general.

Gradient Descent Update Rule: Using the approximated derivative, the update rule in the gradient descent algorithm is:

$$x^{(k+1)} = x^{(k)} - \alpha \left(\frac{g(x^{(k)} + h) - g(x^{(k)} - h)}{2h} \right)$$

where α is the learning rate, dictating step size opposite the gradient.

Conclusion: The central difference formula offers a robust approximation of derivatives, essential for applying gradient descent when derivatives are not analytically accessible, ensuring high accuracy and computational efficiency.