Natural Language Processing
Assignment 1

Original by: Mohammad Taha Majlesi
tahamajlesi@ut.ac.ir, 810101504.

September 30, 2025

Contents

1 The Hessian Matrix: Definition and Properties

1.1 Explanation of Core Concepts

Partial Derivatives and the Gradient For a function of multiple variables, like $f(x_1, x_2, \ldots, x_n)$, a **partial derivative** (e.g., $\frac{\partial f}{\partial x_i}$) measures how the function's output changes when only the variable x_i changes, holding all other variables constant. The **gradient**, denoted as $\nabla f(x)$, is a vector that collects all these partial derivatives. It points in the direction of the steepest ascent of the function at a point x.

Second-Order Derivatives and the Hessian A **second-order partial derivative** (e.g., $\frac{\partial^2 f}{\partial x_j \partial x_i}$) is the derivative of a first-order partial derivative. It tells us how the rate of change itself is changing. The **Hessian matrix**, denoted $H_f(x)$ or $\nabla^2 f(x)$, is a square matrix that organizes all possible second-order partial derivatives of the function. It describes the **local curvature** of the function's surface at a point x. This is crucial for optimization:

- A **positive-definite** Hessian at a critical point indicates a local minimum.
- A **negative-definite** Hessian indicates a local maximum.
- An **indefinite** Hessian indicates a saddle point.

Symmetry of the Hessian According to **Clairaut's theorem**, if the second partial derivatives are continuous, the order of differentiation does not matter (i.e., $\frac{\partial^2 f}{\partial x_i \partial x_j} = \frac{\partial^2 f}{\partial x_j \partial x_i}$). This means the Hessian matrix is always symmetric ($H_f(x) = H_f(x)^T$), a property that simplifies many calculations.

1.2 Formal Derivation

The Hessian matrix of a function $f : \mathbb{R}^n \to \mathbb{R}$ is formally defined as:

$$H_f(x) \triangleq \nabla^2 f(x) = \begin{bmatrix} \frac{\partial^2 f}{\partial x_1^2}(x) & \cdots & \frac{\partial^2 f}{\partial x_1 \partial x_n}(x) \\ \vdots & \ddots & \vdots \\ \frac{\partial^2 f}{\partial x_n \partial x_1}(x) & \cdots & \frac{\partial^2 f}{\partial x_n^2}(x) \end{bmatrix} \quad (1)$$

The i-th column of the Hessian is the collection of second derivatives involving x_i:

$$H_f(x)_{:,i} = \begin{bmatrix} \frac{\partial^2 f}{\partial x_1 \partial x_i}(x) \\ \vdots \\ \frac{\partial^2 f}{\partial x_n \partial x_i}(x) \end{bmatrix} = \nabla \left(\frac{\partial f}{\partial x_i} \right)(x) \quad (2,3)$$

This shows that the i-th column of the Hessian is simply the gradient of the i-th partial derivative of f.

To express this mathematically, we note that the i-th partial derivative, $\frac{\partial f}{\partial x_i}$, is the i-th component of the gradient vector $\nabla f(x)$. We can isolate this component using the dot product with a **standard basis vector** e_i (a vector with a 1 in the i-th position and 0s elsewhere).

$$\frac{\partial f}{\partial x_i}(x) = e_i^T \nabla f(x)$$

Substituting this into our expression for the i-th column (3) gives:

$$H_f(x)_{:,i} = \nabla(e_i^T \nabla f(x)) \quad (4,5)$$

This elegant formula states that the i-th column of the Hessian can be found by taking the gradient of a scalar projection of the original function's gradient. By computing this for each i from 1 to n, we can construct the entire Hessian matrix:

$$H_f(x) = \begin{bmatrix} \nabla(e_1^T \nabla f(x)) & \cdots & \nabla(e_n^T \nabla f(x)) \end{bmatrix} \quad (6)$$

Leveraging the symmetry of the Hessian, we can also define its i-th row as the transpose of its i-th column:

$$H_f(x)_{i,:} = (\nabla(e_i^T \nabla f(x)))^T \quad (7)$$

2 Computational Complexity of the Hessian

2.1 Explanation of Core Concepts

Computational Complexity and Big O Notation **Computational complexity** measures the amount of resources (typically time or memory) an algorithm needs to run. **Big O notation** (e.g., $O(m)$) is used to describe the limiting behavior of this resource usage as the input size grows. It focuses on the dominant term, ignoring constants and lower-order terms.

Computational Graphs and Backpropagation In machine learning, complex functions like neural networks are often visualized as **computational graphs**, where nodes represent operations (e.g., addition, matrix multiplication) and edges represent the flow of data.

- A **forward pass** involves feeding input into the graph and computing the final output value, node by node.
- A **backward pass**, or **backpropagation**, is a clever algorithm that efficiently computes the gradient (∇f) of the output with respect to every node in the graph. A key result is that the time complexity of the backward pass is roughly the same as the forward pass. If a forward pass takes $O(m)$ time, computing the entire gradient also takes $O(m)$ time.

2.2 Complexity Analysis

The task is to find the complexity of computing the full Hessian matrix, $H_f(x)$.

1. **Cost of One Column:** From equation (3), we know that the i-th column of the Hessian, $H_f(x)_{:,i}$, is the gradient of the function $g_i(x) = \frac{\partial f}{\partial x_i}(x)$.
2. **Cost of Gradient Calculation:** Let's assume the forward pass to compute the original function $f(x)$ has a complexity of $O(m)$. The computational graph for each partial derivative $g_i(x)$ will be similar in complexity. Using backpropagation on the graph of $g_i(x)$ to find its gradient, $\nabla g_i(x)$, will therefore also have a complexity of $O(m)$.

3. **Total Cost:** The Hessian has n columns. Since computing each column costs $O(m)$, the total complexity for computing the entire Hessian is the cost per column multiplied by the number of columns:

$$\text{Total Complexity} = n \times O(m) = O(mn)$$

Comparison to a Naive Approach A more naive method would be to compute each of the n^2 elements of the Hessian independently. If each individual second derivative $\frac{\partial^2 f}{\partial x_j \partial x_i}$ took $O(m)$ time to compute, the total time would be $O(mn^2)$. The $O(mn)$ approach is far more efficient because it reuses computations through the structured process of backpropagation on each of the n partial derivative functions.

3 Complexity for Higher-Order Derivatives

3.1 Explanation of Core Concepts

Higher-order derivatives extend the concept of the gradient (1st order) and Hessian (2nd order) to third, fourth, and k-th orders. These are represented by multi-dimensional arrays called **tensors**. While computationally expensive, they can capture even more subtle information about a function's behavior.

3.2 Generalizing the Complexity

We can extrapolate the pattern from our previous analysis.

- **1st Order (Gradient):** We are given its complexity is $O(m) = O(mn^{1-1})$.
- **2nd Order (Hessian):** We found its complexity is $O(mn) = O(mn^{2-1})$. This involves computing n gradients.
- **3rd Order Tensor:** To get the third derivatives, we would need to compute the gradient of each of the n^2 entries of the Hessian matrix. This would involve n^2 backpropagation passes, leading to a complexity of $O(mn^2) = O(mn^{3-1})$.

This reveals a general formula. Computing the tensor of k-th order derivatives has a computational complexity of:

$$\text{Complexity for k-th order} = O(mn^{k-1})$$

4 Efficiently Computing the Diagonal of the Hessian

4.1 Explanation of Core Concepts

Why Compute Only the Diagonal? Calculating the full Hessian ($O(mn)$) can be prohibitively expensive, especially for functions with many input variables (large n), like in deep learning. However, many optimization algorithms (e.g., quasi-Newton methods, Adam) can work effectively with just the diagonal elements of the Hessian. These elements, $\frac{\partial^2 f}{\partial x_j^2}$, provide information about the curvature along each individual dimension, which can be used to adaptively scale learning rates for each parameter. This is a practical trade-off between computational cost and optimization performance.

Derivation via the Chain Rule To find an efficient way to compute $\frac{\partial^2 y_i}{\partial x_j^2}$ (the diagonal elements), we start with the chain rule for the first derivative, where z_k are intermediate variables in the computational graph:

$$\frac{\partial y_i}{\partial x_j} = \sum_{k=1}^{M} \frac{\partial y_i}{\partial z_k} \frac{\partial z_k}{\partial x_j} \quad (11)$$

We then differentiate this expression again with respect to x_j, carefully applying the product rule and chain rule. This leads to the final expression:

$$\frac{\partial^2 y_i}{\partial x_j^2} = \sum_{k=1}^{M} \left[\frac{\partial^2 y_i}{\partial z_k^2} \left(\frac{\partial z_k}{\partial x_j} \right)^2 + \frac{\partial y_i}{\partial z_k} \frac{\partial^2 z_k}{\partial x_j^2} \right] \quad (12)$$

This equation is powerful because it breaks down the desired second derivative into simpler terms that can be computed and reused during a single augmented pass through the computational graph. All terms inside the summation are either first or second derivatives of the simple, elementary functions that make up the graph nodes.

4.2 An Augmented Backpropagation Algorithm

This derivation leads to an efficient, three-step algorithm:

1. **Forward Pass:** Perform a standard forward pass through the computational graph. Compute and store the values of all variables (x_j, z_k, y_i) at each node.
2. **Standard Backward Pass:** Perform a standard backpropagation pass. Compute and store all the first derivatives between connected nodes (e.g., $\frac{\partial y_i}{\partial z_k}$ and $\frac{\partial z_k}{\partial x_j}$). This process is often called "memoization."
3. **Augmented Values Pass:** This is the novel step. During the backward pass (or a separate pass), compute the *second* derivatives of the elementary functions at each node (e.g., $\frac{\partial^2 y_i}{\partial z_k^2}$ and $\frac{\partial^2 z_k}{\partial x_j^2}$). Store these values as well. Finally, use the stored first and second derivatives to compute the diagonal Hessian elements $\frac{\partial^2 y_i}{\partial x_j^2}$ using Equation (12).

This augmented algorithm is highly efficient because it computes everything needed in a structured way, avoiding redundant calculations and the need to ever construct the full Hessian matrix. It effectively "doubles" the amount of information stored per node (both first and second derivatives), but in return provides the Hessian diagonal with much less work than the $O(mn)$ full-Hessian approach.

5 Question 2: Link to Collaborative Notebook

The work for the second question is implemented in codes folder