UNIVERSITAT POLITÈCNICA DE CATALUNYA
BARCELONATECH
Facultat d'Informàtica de Barcelona

FIB

Backpropagation

Theoretical Derivation, Complexity Analysis, Implementation and Numerical Checking

Joan Acero (joan.acero@estudiantat.upc.edu)

Àlex Font (alex.font@estudiantat.upc.edu)

Reviewer: Marc Díaz (marc.diaz.calderon@estudiantat.upc.edu)

Algorithmics for Data Mining (ADM)

April 2025

Professor: Lluís A. Belanche

Academic Year: 2024-2025

Facultat d'Informàtica de Barcelona (FIB)

Universitat Politècnica de Catalunya (UPC)

Abstract

This project addresses the fundamental task of training feed-forward neural networks through the derivation, implementation, and verification of the backpropagation algorithm. We begin by mathematically deriving the backpropagation algorithm for a general multi-layer perceptron (MLP) with arbitrary differentiable activation functions. Moreover, we analyze the complexity of the algorithm in terms of time and space.

Subsequently, a versatile MLP is implemented from scratch using Python and NumPy, capable of performing both K-class classification (with softmax output and cross-entropy loss) and unidimensional regression (with linear output and mean squared error loss), employing ReLU activation functions in hidden layers. The core of this implementation is the backpropagation algorithm, for which detailed gradient derivations for output and hidden layers are presented. To ensure correctness, the analytical gradients computed by our backpropagation implementation are rigorously verified against numerically approximated gradients using the finite difference method. The numerical gradient checking confirmed the accuracy of our implementation, with relative errors consistently below 10^{-7}.

Contents

List of Figures

1 Introduction

The Multi-Layer Perceptron (MLP) stands as a foundational model in machine learning, exhibiting remarkable success across diverse tasks ranging from image recognition to natural language processing. The idea of a perceptron was introduced by Frank Rosenblatt in 1958 [1]. However, the inability of the perceptron to solve non-linearly separable problems, such as the XOR problem, highlighted the necessity for more complex structures.

This limitation spurred the evolution of MLPs, which consist of multiple layers of interconnected neurons, enabling the modeling of intricate, non-linear relationships. The theoretical capabilities of MLPs were solidified by the Universal Approximation Theorem, which asserts that feedforward networks with at least one hidden layer can approximate any continuous function to an arbitrary degree of accuracy, given sufficient neurons and appropriate activation functions [2].

Central to the practical training of MLPs is the backpropagation algorithm, introduced by Rumelhart, Hinton, and Williams in 1986 [3]. This algorithm efficiently computes the gradients of the cost function with respect to all network parameters by systematically applying the chain rule of calculus in a layer-by-layer manner, propagating errors backward from the output layer to the input layer. This approach significantly reduces computational redundancy, making the training of deep networks feasible and practical.

This project aims to provide a thorough exploration of the backpropagation algorithm. The primary objectives are:

1. To derive the backpropagation algorithm for a general feedforward neural network, applicable to arbitrary differentiable activation functions, and to analyze its computational complexity.
2. To implement the backpropagation algorithm from scratch within a custom-built MLP framework using only the NumPy library. This MLP will be designed to handle both classification and regression tasks.
3. To verify the correctness of the implemented backpropagation algorithm through numerical gradient checking, comparing analytically derived gradients with those approximated using finite differences.

The report details the mathematical derivations, the structure of the MLP implementation, the specifics of the backpropagation code for different task types, and the methodology and results of the numerical gradient checking procedure.

2 Background

This section introduces the key concepts necessary for understanding the development of this project. While the objective is not to explore these topics in exhaustive detail, we aim to provide sufficient context to support comprehension of the subsequent sections. The background material presented here is primarily adapted from Ian Goodfellow's Deep Learning textbook [4].

2.1 Multi-Layer Perceptron

A **perceptron** is an abstract computing unit that receives an **input** vector, combines it with a vector of local parameters called **weights**, and often a **bias** term, then applies a non-linear transformation to produce an **output** scalar [1]. Perceptrons are also referred to as *neurons* or *units*.

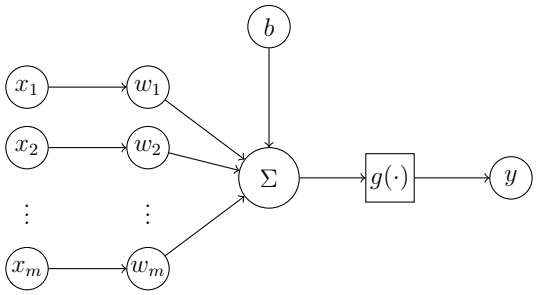

Figure 1: Perceptron computing $y = g(w^\top x + b)$.

Mathematically, given an input vector $x \in \mathbb{R}^n$, a neuron computes:

$$y = g(w^\top x + b)$$

where $w \in \mathbb{R}^n$ is the weight vector, $b \in \mathbb{R}$ is the bias, and $g : \mathbb{R} \to \mathbb{R}$ is a non-linear function (usually denoted activation function), such as the sigmoid, tanh, or Rectified Linear Unit (ReLU) function. The activation function introduces non-linearity, enabling the network to learn complex patterns beyond linear combinations of inputs. Figure 1 depicts a perceptron.

A set of neurons that are not connected and take the same input vector forms a **layer**. For m neurons in a layer, each with weights w_i and bias b_i, the layer collectively computes:

$$z = W^\top x + b, \quad h = g(z)$$

where:

- $W \in \mathbb{R}^{n \times m}$ is the weight matrix (each column corresponds to a neuron's weights),
- $b \in \mathbb{R}^m$ is the bias vector,
- g is applied element-wise to the vector z.

Stacking multiple layers of this kind leads to a **feedforward neural network**, also known as an **MLP**. An MLP is composed of an input layer, one or more hidden layers, and an output layer (see Figure 2). Each layer receives the output from the previous layer as input and applies a linear transformation followed by a non-linear activation function.

The layers in a neural network are typically grouped into three types:

- Input layer: This is where the raw input features (denoted x) enter the network.
- Hidden layers: These are the intermediate layers between input and output. Each neuron in a hidden layer computes a weighted sum of its inputs, adds a bias, and applies a nonlinear activation function. They're called "hidden" because their outputs are not directly observed.

- Output layer: This layer produces the final result of the network, for example, a class label in classification tasks or a numeric prediction in regression problems.

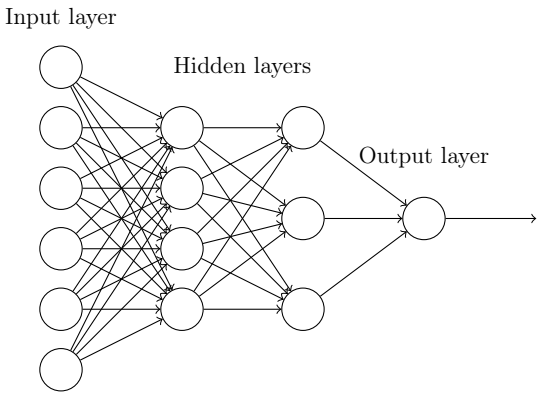

Figure 2: Structure of a multilayer perceptron (MLP) with an input layer, two hidden layers, and one output layer.

Formally, for an MLP with L layers (not counting the input layer), we define:

$$h^{(0)} = x, \quad z^{(l)} = (W^{(l)})^\top h^{(l-1)} + b^{(l)}, \quad h^{(l)} = g(z^{(l)}) \quad \text{for } l = 1, \ldots, L$$

Here:

- $h^{(l)}$ is the vector of activations at layer l,
- $W^{(l)}$ and $b^{(l)}$ are the weights and biases at layer l,
- g is the activation function. We could also consider the case where the activation function is not the same for every layer.

The output of the final layer $h^{(L)}$ is the outcome of the network. The structure of an MLP allows it to approximate a broad class of functions and is especially powerful when trained with large datasets. In practice, the specific number of layers, neurons per layer, and activation functions are chosen based on the problem domain and are considered part of the model architecture.

Note that the activation function of the output layer is often chosen depending on the task:

- For **regression**, the output is often linear: $h^{(L)} = z^{(L)}$
- For **classification**, we often take a softmax activation: $h_j^{(L)} = e^{z_j^{(L)}} / \sum_k e^{z_k^{(L)}}$

These choices ensure the output is in the appropriate range (e.g., probabilities between 0 and 1 for classification).

2.1.1 MLP Training

To train an MLP, we define a **cost function** (also called a loss function) that quantifies the discrepancy between the network's predictions and the actual target values. The goal of training is to find the network parameters (weights and biases) that minimize this cost function over the training dataset.

Let the dataset be $\{(x^{(i)}, y^{(i)})\}_{i=1}^{n}$, where $x^{(i)}$ denotes an input and $y^{(i)}$ the corresponding target output. A common choice of cost function in regression problems is the **mean squared error (MSE)** [5]:

$$C = \frac{1}{n} \sum_{i=1}^{n} \left\| h^{(L)}(x^{(i)}) - y^{(i)} \right\|^2$$

Here, $h^{(L)}(x^{(i)})$ is the output of the network (at the final layer L) when given input $x^{(i)}$.

For classification tasks, especially binary or multi-class classification, other cost functions such as the **cross-entropy loss** are typically used [5]. The choice of cost function depends on the nature of the problem and the activation function used in the output layer.

To minimize the cost function, we use an optimization algorithm called **gradient descent**. This method updates the network parameters in the direction of the negative gradient of the cost function, which points toward the steepest descent. For a given parameter θ (which could be a weight or bias), the update rule is:

$$\theta \leftarrow \theta - \eta \frac{\partial C}{\partial \theta}$$

where $\eta > 0$ is the **learning rate**, a small constant that controls the step size of each update. This process is repeated iteratively across many epochs of training.

In practice, computing the gradient over the entire dataset at each iteration (known as **batch gradient descent**) can be computationally expensive. Instead, we often use **stochastic gradient descent (SGD)**, which updates the parameters using the gradient computed on a single training example or a small mini-batch. This results in faster, more frequent updates and introduces noise that can help escape local minima or saddle points.

To compute these gradients efficiently in a multilayer network, we use the **backpropagation algorithm**, which we describe next.

Moreover, while weights can be initialized randomly, specific techniques, such as Xavier or He initialization (which we used in our custom MLP implementation), often yield better performance and faster convergence [6].

3 Backpropagation

Backpropagation is a general algorithm used to compute the derivative of a function. While it is most commonly associated with training neural networks, it can be applied to any differentiable function. The idea behind backpropagation is to compute the gradient by recursively applying the chain rule, propagating derivatives from the output back to the inputs through intermediate variables. This allows for efficient calculation of the gradient, even in complex, high-dimensional functions, by reusing partial derivatives.

In the context of feed-forward neural networks, the backpropagation algorithm computes the derivatives of the cost function with respect to the weights and biases needed to apply gradient descent.

We will now derive the backpropagation algorithm for a general feed-forward neural network with arbitrary differentiable activation functions.

Consider the cost function to be the average of the cost for every data point x. The derivation works the same for a general cost function. For instance if we take the cost function to be the mean squared error we get:

$$C = \frac{1}{|X|}\sum_{x\in X} C_x, \quad \text{where} \quad C_x = \left\|h^{(L)}(x) - y\right\|^2 = \sum_{j=1}^{n_L}\left(h_j^{(L)} - y_j\right)^2$$

Then, we put some names to intermediate values that we will use to compute the derivatives.

$$h^{(l)} = g(z^{(l)}) \ , \quad z^{(l)} = (W^{(l)})^\top h^{(l-1)} + b^{(l)}$$

It will be useful to have this expressions not only in vector notation.

$$h_i^{(l)} = g(z_i^{(l)}) \ , \quad z_i^{(l)} = \sum_{j=1}^{n_{l-1}} w_{ji}^{(l)} h_j^{(l-1)} + b_i^{(l)}$$

We want to compute the derivatives of the cost function with respect to the weights and biases, that is:

$$\nabla C = (\frac{\partial C}{\partial w}, \frac{\partial C}{\partial b})^\top$$

$$\frac{\partial C}{\partial w_{ij}^{(l)}} = \frac{1}{|X|}\sum_{x\in X}\frac{\partial C_x}{\partial w_{ij}^{(l)}} \ , \qquad \frac{\partial C}{\partial b_i^{(l)}} = \frac{1}{|X|}\sum_{x\in X}\frac{\partial C_x}{\partial b_i^{(l)}}$$

Applying the chain rule we can express the derivatives of the cost with respect to the weights w and biases b as a product involving the derivative with respect to the activations h. We can see how they interact in the computational graph, as sketched in figure 3.

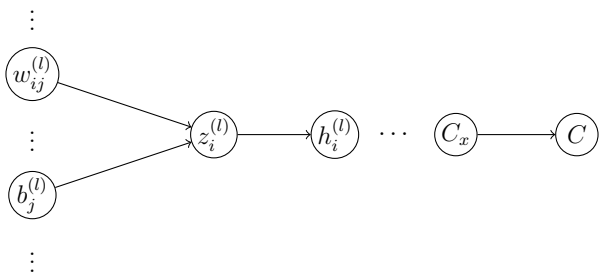

Figure 3: Computational graph of a general feed-forward neural network. Each node represents a value in the forward pass, and arrows indicate dependence, which informs how the chain rule is applied in reverse.

$$\frac{\partial C_x}{\partial w_{ij}^{(l)}} = \frac{\partial z_i^{(l)}}{\partial w_{ij}^{(l)}}\frac{\partial h_i^{(l)}}{\partial z_i^{(l)}}\frac{\partial C_x}{\partial h_i^{(l)}} \ , \qquad \frac{\partial C_x}{\partial b_j^{(l)}} = \frac{\partial z_i^{(l)}}{\partial b_j^{(l)}}\frac{\partial h_i^{(l)}}{\partial z_i^{(l)}}\frac{\partial C_x}{\partial h_i^{(l)}}$$

Similarly, to compute this derivative with respect to the activations, we again apply the chain rule to get a recursive expression, where n_l is the number of neurons in layer l. We can see this dependence in the computational graph represented in figure 4.

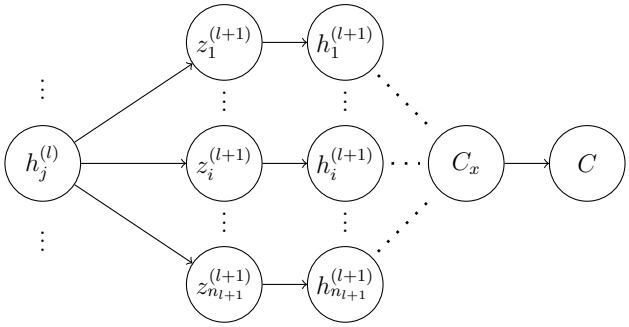

Figure 4: Dependence of $h_i^{(l+1)}$ with respect to $h_j^{(l)}$ in the computational graph of a general feed-forward neural network.

$$\frac{\partial C_x}{\partial h_i^{(L)}} = 2(h_i^{(L)} - y_i)$$
$$\frac{\partial C_x}{\partial h_j^{(l)}} = \sum_{i=1}^{n_{l+1}} \frac{\partial z_i^{(l+1)}}{\partial h_j^{(l)}} \frac{\partial h_i^{(l+1)}}{\partial z_i^{(l+1)}} \frac{\partial C_x}{\partial h_i^{(l+1)}} \qquad \text{for layer } l < L$$

With this recursive expression for the partial derivatives with respect to the activations and all derivatives we need expressed in terms of derivatives with respect to the activations, we can make a recursive algorithm to compute the gradient. We also need to know the following derivatives:

$$\frac{\partial z_i^{(l)}}{\partial w_{ij}^{(l)}} = h_j^{(l-1)} \ , \quad \frac{\partial z_i^{(l)}}{\partial b_i^{(l)}} = 1 \ , \quad \frac{\partial h_i^{(l)}}{\partial z_i^{(l)}} = g'(z_i^{(l)}) \ , \quad \frac{\partial z_i^{(l)}}{\partial h_j^{(l-1)}} = w_{ij}^{(l)} \ ,$$

Then, a natural recursive algorithm emerges from this expression. However, we can observe that some partial derivatives of intermediate layers appear in multiple terms. If we were to compute each derivative directly using the recursive form, we would end up repeating many calculations. To make this process more efficient, we apply dynamic programming: we compute the derivatives in a specific order, moving from the output layer back to the input layer. This ensures that all necessary intermediate results are available when needed and can be reused. This backward flow of computations, propagating the derivatives from the output all the way back to inputs, is what gives the algorithm its name.

To express the backpropagation algorithm more compactly and enable efficient numerical implementation, we now rewrite the scalar derivatives in vector notation, following the same notation introduced earlier. The forward pass remains:

$$z^{(l)} = (W^{(l)})^\top h^{(l-l)} + b^{(l)} \ , \quad h^{(l)} = g(z^{(l)})$$

where g acts element-wise.
The recursive derivative with respect to the activations at layer $l < L$ is:

$$\frac{\partial C_x}{\partial h^{(l)}} = (W^{(l+1)})^\top \left(\frac{\partial C_x}{\partial h^{(l+1)}} \odot g'(z^{(l+1)}) \right)$$

where $\odot$ means the element-wise product, also called Hadamard product.
Using these, we can compute the gradients with respect to the weights and biases as:

$$\frac{\partial C_x}{\partial W^{(l)}} = h^{(l-1)} \cdot \left(\frac{\partial C_x}{\partial h^{(l)}} \odot g'(z^{(l)}) \right)^\top \ , \quad \frac{\partial C_x}{\partial b^{(l)}} = \frac{\partial C_x}{\partial h^{(l)}} \odot g'(z^{(l)})$$

These expressions naturally lead to a recursive algorithm where all gradients are expressed in terms of partial derivatives with respect to the activations $h^{(l)}$, which are computed layer by layer from the output back to the input. Moreover, this formulation makes the algorithm amenable to efficient implementation using vectorized numerical libraries such as NumPy, as we do in our implementation 5.

Algorithm 1: Backpropagation Algorithm.
We define $dh^{(l)} := \partial C_x / \partial h^{(l)}$, $dW^{(l)} := \partial C_x / \partial W^{(l)}$, $db^{(l)} := \partial C_x / \partial b^{(l)}$.

Input: Data x, real values y, weights $W^{(l)}$, biases $b^{(l)}$, activation function g and its derivative g'
Output: Gradients $dW^{(l)}, db^{(l)}$ for each layer l

Forward pass:
for $l \leftarrow 1$ **to** L **do**
  $z^{(l)} \leftarrow (W^{(l)})^\top h^{(l-1)} + b^{(l)}$
  $h^{(l)} \leftarrow g(z^{(l)})$

Initialize derivative at output layer:
$dh^{(L)}$ `// This depends on the specific cost function`

Backward pass:
for $l \leftarrow L$ **to** 1 **do**
  $v^{(l)} \leftarrow dh^{(l)} \odot g'(z^{(l)})$
  $dW^{(l)} \leftarrow h^{(l-1)} (v^{(l)})^\top$
  $db^{(l)} \leftarrow v^{(l)}$
  if $l > 1$ **then**
    $dh^{(l-1)} \leftarrow W^{(l)} v^{(l)}$

4 Computational Complexity

Now we will analyze the computational complexity of backpropagation in terms of the number of layers, neurons per layer, and batch size.
Let:

- L: number of layers (excluding input layer)
- n_l: number of neurons in layer l
- $d = n_0$: input dimension
- m: number of data points in the dataset (or minibatch size)

We assume the activation function g and its derivative g' take constant time per neuron. In the following, we state the complexities for a single data sample. The total cost for a minibatch of size m will then scale linearly with m, unless otherwise noted.

4.1 Forward Pass Complexity

For each layer $l = 1, \ldots, L$:

- Linear combination: $z^{(l)} = (W^{(l)})^\top h^{(l-1)} + b^{(l)}$ $\mathcal{O}(n_{l-1} \cdot n_l)$
- Activation: $h^{(l)} = g(z^{(l)})$ $\mathcal{O}(n_l)$

Total cost of the forward pass:

$$\mathcal{O}\left(\sum_{l=1}^{L} n_{l-1} \cdot n_l\right)$$

4.2 Backward Pass Complexity

For each layer $l = L, \ldots, 1$:

- Elementwise gradient: $v^{(l)} = dh^{(l)} \odot g'(z^{(l)})$ $\mathcal{O}(n_l)$
- Gradients w.r.t. weights: $dW^{(l)} = h^{(l-1)}(v^{(l)})^\top$ $\mathcal{O}(n_{l-1} \cdot n_l)$
- Gradients w.r.t. biases: $db^{(l)} = v^{(l)}$ $\mathcal{O}(n_l)$
- Gradient propagation: $dh^{(l-1)} = W^{(l)} v^{(l)}$ $\mathcal{O}(n_{l-1} \cdot n_l)$

Total cost of the backward pass:

$$\mathcal{O}\left(\sum_{l=1}^{L} n_{l-1} \cdot n_l\right)$$

For a dataset of minibatch of size m we need to scale these quantities by a factor of m, so the total complexity becomes

$$\mathcal{O}\left(m \cdot \sum_{l=1}^{L} n_{l-1} \cdot n_l\right)$$

Parallelism and Practical Efficiency. Although the theoretical computational cost scales linearly with the batch size m, libraries such as NumPy, TensorFlow, or PyTorch exploit vectorized operations and hardware-level parallelism to execute these computations efficiently. Rather than performing m separate matrix-vector operations, the forward and backward passes are expressed as matrix-matrix multiplications, enabling simultaneous computation across all samples.

Furthermore, parallelism extends beyond the batch dimension: element-wise operations (like applying activation functions) and operations across neurons within a layer are also parallelized. While this reduces runtime significantly in practice, the total number of arithmetic operations remains $\mathcal{O}(m \cdot n_{l-1} \cdot n_l)$ per layer.

4.3 Space Complexity

Backpropagation stores intermediate values during the forward pass to avoid redundant computation. Specifically, it stores the activations $h^{(l)}$ and pre-activations $z^{(l)}$ for each layer, which are required to compute gradients during the backward pass. In addition, temporary variables such as $dh^{(l)}$ and $v^{(l)}$ are maintained during backpropagation.

For a single data sample, this requires:

$$\mathcal{O}\left(\sum_{l=1}^{L} n_l\right)$$

If a batch of m samples is processed in parallel, each sample requires separate storage for these values, resulting in:

$$\mathcal{O}\left(m \cdot \sum_{l=1}^{L} n_l\right)$$

Note. Model parameters $W^{(l)}$, $b^{(l)}$ and their corresponding gradients $dW^{(l)}$, $db^{(l)}$ are also stored, requiring $\mathcal{O}\left(\sum_{l=1}^{L} n_{l-1} \cdot n_l\right)$ space. However, this cost is independent of the batch size m, and it does not scale with the number of data samples. It can be considered a fixed overhead, separate from the memory cost that grows linearly with m.

Optional memory optimizations. The space complexity analysis above assumes that all intermediate values are stored during the forward pass. In practice, this memory usage can be reduced using techniques such as *activation checkpointing*, where only a subset of activations is stored, and the rest are recomputed during backpropagation. This reduces memory at the cost of additional computation, and is especially useful in deep networks where memory is a limiting factor.

5 Implementation

We developed an entire custom MLP class from scratch by using only the numpy library to ensure a fundamental understanding of its operational mechanics. The implementation is versatile, capable of performing both **regression** (unidimensional) and **classification** tasks, and allowing users to define the **number of hidden layers** and the **number of neurons within each**. All hidden layers consistently use **ReLU** activation functions. For classification, the network employs a softmax activation in the output layer paired with the cross-entropy loss (which is equivalent to the negative log-likelihood for the implied multinomial distribution). At the same time, regression tasks use a linear activation function at the output with Mean Squared Error as the loss function. The code is attached in Appendix A.

To implement the backpropagation algorithm we derived the gradients of all parameters (weights $W^{(l)}$ and biases $b^{(l)}$) over the cost function C, both for the classification and the regression tasks. The computations can be divided into the output layer and the hidden layers. Below, we present the derivations that led to our implementation of the backpropagation algorithm.

5.1 Output Layer Gradients

This section details the computation of gradients for the parameters of the output layer, denoted as layer L. The core idea is to first determine the gradient of the cost function C with respect to the pre-activations $\mathbf{Z}^{(L)}$ of this layer. This intermediate gradient, denoted $\delta^{(L)}_{\text{batch}} = \frac{\partial C}{\partial \mathbf{Z}^{(L)}_{\text{batch}}}$, encapsulates the error signal at the output layer's input. The specific form of $\delta^{(L)}_{\text{batch}}$ depends on the task (classification or regression), the output activation function, and the chosen cost function.

The derivation proceeds in three main parts:

1. **K-class Classification Case (Subsection 5.1.1):** We derive $\delta^{(L)}_{\text{batch}}$ for a K-class classification task, employing a softmax activation function in the output layer and a cross-entropy loss function.
2. **Uni-Dimensional Regression Case (Subsection 5.1.2):** We derive $\delta^{(L)}_{\text{batch}}$ for a uni-dimensional regression task, using a linear activation function and a squared error loss function.
3. **Gradients for Output Layer Parameters $W^{(L)}$ and $\mathbf{b}^{(L)}$ (Subsection 5.1.3):** Finally, we present the general formulas to compute the gradients $\frac{\partial C}{\partial W^{(L)}}$ and $\frac{\partial C}{\partial \mathbf{b}^{(L)}}$. These formulas utilize the task-specific $\delta^{(L)}_{\text{batch}}$ (derived in the preceding subsections) and the activations from the previous layer $\mathbf{H}^{(L-1)}_{\text{batch}}$.

Throughout these derivations, we first consider a single data point x with cost C_x, and then extend the results to a mini-batch of n samples where the total cost $C = \frac{1}{n}\sum_{i=1}^{n} C_{x_i}$ is the average per-sample cost.

5.1.1 K-class classification case

Consider that the MLP is being used for K-class classification, meaning the output layer L has $n_L = K$ neurons. Figure 5 shows the structure of the output layer, where:

- $\mathbf{z}^{(L)} = (z_1^{(L)}, \ldots, z_K^{(L)})$ are the pre-activations of the output layer L.
- $\mathbf{h}^{(L)} = (h_1^{(L)}, \ldots, h_K^{(L)})$ are the activations (probabilities) from the softmax function $g(\cdot)$:

$$h_i^{(L)} = g(z_i^{(L)}) = \frac{e^{z_i^{(L)}}}{\sum_{m=1}^{K} e^{z_m^{(L)}}}$$

- $\mathbf{y} = (y_1, \ldots, y_K)$ is the one-hot encoded true label vector, where $y_c = 1$ for the true class c, and $y_k = 0$ for $k \neq c$. Thus, $\sum_{m=1}^{K} y_m = 1$.
- C_x is the cost function for a single data point x, which we set to the negative log-likelihood for mathematical convenience.

$$C_x = -\log P(y \mid x) = -\log \left(\prod_{m=1}^{K} (h_m^{(L)})^{y_m} \right) = -\sum_{m=1}^{K} y_m \log(h_m^{(L)})$$

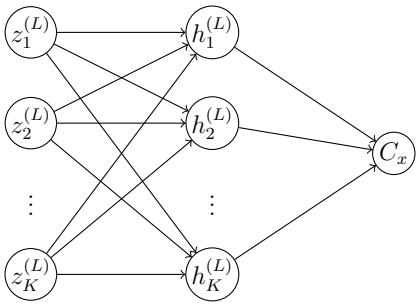

Figure 5: Output layer L structure for classification. Each activation $h_j^{(L)}$ depends on all pre-activations $z_i^{(L)}$ (due to the softmax denominator), and the cost C_x depends on all $h_i^{(L)}$.

Our objective is to compute the gradients of the cost function with respect to the network parameters $(W^{(L)}, \mathbf{b}^{(L)})$. We begin by finding the gradient of C_x with respect to the pre-activations $z_{x,j}^{(L)}$.

$$\frac{\partial C_x}{\partial z_j^{(L)}} = \sum_{i=1}^{K} \frac{\partial C_x}{\partial h_i^{(L)}} \frac{\partial h_i^{(L)}}{\partial z_j^{(L)}} \tag{1}$$

We need to compute two parts: $\frac{\partial C_x}{\partial h_i^{(L)}}$ and $\frac{\partial h_i^{(L)}}{\partial z_j^{(L)}}$.

Step 1: Derivative of the Loss with respect to Softmax $\frac{\partial C_x}{\partial h_i^{(L)}}$

$$\begin{aligned}
\frac{\partial C_x}{\partial h_i^{(L)}} &= \frac{\partial}{\partial h_i^{(L)}}\left(-\sum_{m=1}^{K} y_m \log(h_m^{(L)})\right) \\
&= -y_i \frac{\partial}{\partial h_i^{(L)}}(\log(h_i^{(L)})) - \sum_{m\neq i} y_m \frac{\partial}{\partial h_i^{(L)}}(\log(h_m^{(L)})) \\
&= -y_i \cdot \frac{1}{h_i^{(L)}} - \sum_{m\neq i} y_m \cdot 0 \\
\frac{\partial \mathbf{C_x}}{\partial \mathbf{h_i^{(L)}}} &= -\frac{\mathbf{y_i}}{\mathbf{h_i^{(L)}}}
\end{aligned} \tag{2}$$

Step 2: Derivative of Softmax with respect to the pre-activations $\frac{\partial \mathbf{h_i^{(L)}}}{\partial \mathbf{z_j^{(L)}}}$

To do it, let $S = \sum_{m=1}^{K} e^{z_m^{(L)}}$. Then $h_i^{(L)} = \frac{e^{z_i^{(L)}}}{S}$.

Case 1: $i \neq j$

$$\begin{aligned}
\frac{\partial h_i^{(L)}}{\partial z_j^{(L)}} &= \frac{\partial}{\partial z_j^{(L)}}\left(\frac{e^{z_i^{(L)}}}{S}\right) = e^{z_i^{(L)}} \frac{\partial}{\partial z_j^{(L)}}(S^{-1}) = e^{z_i^{(L)}}(-1 \cdot S^{-2}) \frac{\partial S}{\partial z_j^{(L)}} \\
&= e^{z_i^{(L)}}(-S^{-2}) e^{z_j^{(L)}} = -\frac{e^{z_i^{(L)}} e^{z_j^{(L)}}}{S^2} = -\left(\frac{e^{z_i^{(L)}}}{S}\right)\left(\frac{e^{z_j^{(L)}}}{S}\right) \\
&= -\mathbf{h_i^{(L)}} \mathbf{h_j^{(L)}}
\end{aligned}$$

Case 2: $i = j$

$$\begin{aligned}
\frac{\partial h_i^{(L)}}{\partial z_i^{(L)}} &= \frac{\partial}{\partial z_i^{(L)}}\left(\frac{e^{z_i^{(L)}}}{S}\right) = \frac{(\frac{\partial}{\partial z_i^{(L)}} e^{z_i^{(L)}})S - e^{z_i^{(L)}}(\frac{\partial S}{\partial z_i^{(L)}})}{S^2} \\
&= \frac{e^{z_i^{(L)}} S - e^{z_i^{(L)}} e^{z_i^{(L)}}}{S^2} = \frac{e^{z_i^{(L)}}(S - e^{z_i^{(L)}})}{S^2} \\
&= \frac{e^{z_i^{(L)}}}{S} \cdot \frac{S - e^{z_i^{(L)}}}{S} = h_i^{(L)}\left(1 - \frac{e^{z_i^{(L)}}}{S}\right) \\
&= \mathbf{h_i^{(L)}}(\mathbf{1} - \mathbf{h_i^{(L)}})
\end{aligned}$$

Combining these two cases:

$$\frac{\partial h_i^{(L)}}{\partial z_j^{(L)}} = g'(z_j^{(L)})_i = \begin{cases} h_i^{(L)}(1 - h_i^{(L)}) & \text{if } i = j \\ -h_i^{(L)} h_j^{(L)} & \text{if } i \neq j \end{cases} \tag{3}$$

Step 3: Use the Chain Rule by substituting equations (2) and (3) into (1) to compute $\frac{\partial C_x}{\partial z_j^{(L)}}$.

$$\begin{aligned}\frac{\partial C_x}{\partial z_j^{(L)}} &= \sum_{i=1}^{K} \frac{\partial C_x}{\partial h_i^{(L)}} \frac{\partial h_i^{(L)}}{\partial z_j^{(L)}} = \left(\frac{\partial C_x}{\partial h_j^{(L)}} \frac{\partial h_j^{(L)}}{\partial z_j^{(L)}}\right) + \sum_{i \neq j} \left(\frac{\partial C_x}{\partial h_i^{(L)}} \frac{\partial h_i^{(L)}}{\partial z_j^{(L)}}\right) \\ &= \left(-\frac{y_j}{h_j^{(L)}}\right)(h_j^{(L)}(1 - h_j^{(L)})) + \sum_{i \neq j} \left(-\frac{y_i}{h_i^{(L)}}\right)(-h_i^{(L)} h_j^{(L)}) \\ &= -y_j(1 - h_j^{(L)}) + \sum_{i \neq j} y_i h_j^{(L)} = -y_j + y_j h_j^{(L)} + h_j^{(L)} \sum_{i \neq j} y_i \\ &= -y_j + h_j^{(L)} \left(y_j + \sum_{i \neq j} y_i\right) = -y_j + h_j^{(L)} \sum_{i=1}^{K} y_i\end{aligned}$$

Since $\mathbf{y}$ is a one-hot vector, $\sum_{i=1}^{K} y_i = 1$. Therefore, for a single data point x:

$$\frac{\partial C_x}{\partial z_j^{(L)}} = -y_j + h_j^{(L)}(1) = h_j^{(L)} - y_j$$

In vector notation for a single data point x:

$$\frac{\partial C_x}{\partial \mathbf{z}^{(L)}} = \mathbf{h}^{(L)} - \mathbf{y} \tag{4}$$

For a batch of n data points, the gradient of the total cost $C = \frac{1}{n} \sum_x C_x$ with respect to the batch pre-activations $\mathbf{Z}_{batch}^{(L)}$ is $\frac{1}{n} \sum_x (\mathbf{h}^{(L)}(x) - \mathbf{y}(x))$. In matrix form where $\mathbf{H}_{batch}^{(L)}$ contains activations for all data points in the batch and $\mathbf{Y}_{batch}$ contains the true labels:

$$\frac{\partial C}{\partial \mathbf{Z}_{batch}^{(L)}} = \frac{1}{n}(\mathbf{H}_{batch}^{(L)} - \mathbf{Y}_{batch}) \tag{5}$$

5.1.2 Uni-Dimensional Regression Case

Consider that the MLP is being used for a regression task where the output is a single continuous value (i.e., unidimensional output, meaning the output layer L has $n_L = 1$ neuron). We first derive the gradients for a single data point x.

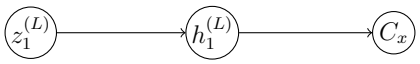

Figure 6: Output layer structure for uni-dimensional regression for a single data point x. The cost C_x depends on the single activation $h_1^{(L)}$, which in turn depends on the pre-activation $z_1^{(L)}$.

- For a single data point x, let $z_1^{(L)}$ be the scalar pre-activation of the single output unit in the output layer L.
- $h_1^{(L)}$ is the scalar activation from the linear activation function $g(\cdot)$: $h_1^{(L)} = g(z_1^{(L)}) = z_1^{(L)}$.

- y_1 is the true scalar continuous target value for this output unit.
- The cost for this data point x, C_x, is the squared error for this output unit:

$$C_x = (h_1^{(L)} - y_1)^2$$

Our goal is to compute $\frac{\partial C_x}{\partial z_1^{(L)}}$. Using the chain rule:

$$\frac{\partial C_x}{\partial z_1^{(L)}} = \frac{\partial C_x}{\partial h_1^{(L)}} \frac{\partial h_1^{(L)}}{\partial z_1^{(L)}} \tag{6}$$

We need to compute two parts: $\frac{\partial C_x}{\partial h_1^{(L)}}$ and $\frac{\partial h_1^{(L)}}{\partial z_1^{(L)}}$.

Step 1: Derivative of the cost function with respect to the activation: $\frac{\partial C_x}{\partial h_1^{(L)}}$

$$\begin{aligned}
\frac{\partial C_x}{\partial h_1^{(L)}} &= \frac{\partial}{\partial h_1^{(L)}} \left((h_1^{(L)} - y_1)^2 \right) \\
&= 2(h_1^{(L)} - y_1) \cdot \frac{\partial}{\partial h_1^{(L)}} (h_1^{(L)} - y_1) \\
&= 2(h_1^{(L)} - y_1) \cdot 1 \\
&\boxed{\frac{\partial \mathbf{C_x}}{\partial \mathbf{h_1^{(L)}}} = \mathbf{2(h_1^{(L)} - y_1)}}
\end{aligned} \tag{7}$$

Step 2: Derivative of the activation function with respect to the pre-activation $\frac{\partial h_1^{(L)}}{\partial z_1^{(L)}}$ Given the linear activation function $h_1^{(L)} = g(z_1^{(L)}) = z_1^{(L)}$:

$$\frac{\partial h_1^{(L)}}{\partial z_1^{(L)}} = \frac{\partial}{\partial z_1^{(L)}} (z_1^{(L)}) = g'(z_1^{(L)}) = 1$$

$$\boxed{\frac{\partial \mathbf{h_1^{(L)}}}{\partial \mathbf{z_1^{(L)}}} = \mathbf{1}} \tag{8}$$

Step 3: Use the Chain Rule by substituting equations (7) and (8) into (6) to compute $\frac{\partial C_x}{\partial z_1^{(L)}}$.

$$\frac{\partial C_x}{\partial z_1^{(L)}} = \Big(2(h_1^{(L)} - y_1) \Big) \cdot 1 = 2(h_1^{(L)} - y_1)$$

In vector notation (where $\mathbf{z}^{(L)} = (z_1^{(L)})$, $\mathbf{h}^{(L)} = (h_1^{(L)})$, $\mathbf{y} = (y_1)$ are single-element vectors for the output layer L of a single data point):

$$\boxed{\frac{\partial C_x}{\partial \mathbf{z}^{(L)}} = 2(\mathbf{h}^{(L)} - \mathbf{y})} \tag{9}$$

For a batch of n data points, when the total cost C is the average squared error $C = \frac{1}{n}\sum_x (h_1^{(L)}(x) - y_1(x))^2$, the gradient of this total cost with respect to the batch pre-activations $\mathbf{Z}_{\text{batch}}^{(L)}$ is effectively $\frac{1}{n}\sum_x [2 \cdot (h_1^{(L)}(x) - y_1(x))]$.

In matrix form, where $\mathbf{H}_{\text{batch}}^{(L)}$ contains the scalar activations $h_1^{(L)}(x)$ for all data points in the batch and $\mathbf{Y}_{\text{batch}}$ contains the scalar true targets $y_1(x)$:

$$\frac{\partial C}{\partial \mathbf{Z}_{\text{batch}}^{(L)}} = \frac{2}{n}(\mathbf{H}_{\text{batch}}^{(L)} - \mathbf{Y}_{\text{batch}}) \tag{10}$$

5.1.3 Gradients for Output Layer Parameters ($W^{(L)}, \mathbf{b}^{(L)}$) from $\delta_{\text{batch}}^{(L)}$

Let $\delta_{\text{batch}}^{(L)}$ denote the gradient $\frac{\partial C}{\partial \mathbf{Z}_{\text{batch}}^{(L)}}$, which has been derived for the specific task (either $\delta_{\text{batch, class}}^{(L)}$ or $\delta_{\text{batch, reg}}^{(L)}$). This matrix has dimensions $n_L \times n$. Let $\mathbf{H}_{\text{batch}}^{(L-1)}$ be the $n_{L-1} \times n$ matrix of activations from layer $L-1$. The pre-activation $Z_{j,i}^{(L)}$ (for neuron j of layer L, sample i) is $Z_{j,i}^{(L)} = \sum_{k=1}^{n_{L-1}} W_{jk}^{(L)} H_{k,i}^{(L-1)} + b_j^{(L)}$.

Gradient w.r.t. Weights $W^{(L)}$ To find $\frac{\partial C}{\partial W_{pq}^{(L)}}$ (for weight connecting neuron q of layer $L-1$ to neuron p of layer L):

$$\frac{\partial C}{\partial W_{pq}^{(L)}} = \sum_{i=1}^{n} \frac{\partial C}{\partial Z_{p,i}^{(L)}} \frac{\partial Z_{p,i}^{(L)}}{\partial W_{pq}^{(L)}}$$

We know $\frac{\partial C}{\partial Z_{p,i}^{(L)}} = (\delta_{\text{batch}}^{(L)})_{p,i}$ and $\frac{\partial Z_{p,i}^{(L)}}{\partial W_{pq}^{(L)}} = H_{q,i}^{(L-1)}$.

$$\frac{\partial C}{\partial W_{pq}^{(L)}} = \sum_{i=1}^{n} (\delta_{\text{batch}}^{(L)})_{p,i} H_{q,i}^{(L-1)}$$

In matrix form:

$$\frac{\partial C}{\partial W^{(L)}} = \delta_{\text{batch}}^{(L)} (\mathbf{H}_{\text{batch}}^{(L-1)})^T \tag{11}$$

Gradient w.r.t. Biases $\mathbf{b}^{(L)}$ For the bias $b_p^{(L)}$ of the p-th neuron in the output layer:

$$\frac{\partial C}{\partial b_p^{(L)}} = \sum_{i=1}^{n} \frac{\partial C}{\partial Z_{p,i}^{(L)}} \frac{\partial Z_{p,i}^{(L)}}{\partial b_p^{(L)}}$$

With $\frac{\partial C}{\partial Z_{p,i}^{(L)}} = (\delta_{\text{batch}}^{(L)})_{p,i}$ and $\frac{\partial Z_{p,i}^{(L)}}{\partial b_p^{(L)}} = 1$.

$$\frac{\partial C}{\partial b_p^{(L)}} = \sum_{i=1}^{n} (\delta_{\text{batch}}^{(L)})_{p,i} \cdot 1$$

For the entire bias vector $\mathbf{b}^{(L)}$ (an $n_L \times 1$ vector):

$$\left(\frac{\partial C}{\partial \mathbf{b}^{(L)}}\right)_p = \sum_{i=1}^{n} (\delta_{\text{batch}}^{(L)})_{p,i} \quad \text{for each } p \in \{1, \ldots, n_L\} \tag{12}$$

Since $\delta_{\text{batch}}^{(L)}$ is defined as the gradient of the average batch cost, it already incorporates the $\frac{1}{n}$ scaling factor.

5.2 Hidden Layer Gradients

For any hidden layer l (where $1 \leq l < L$), the gradients of the average batch cost C with respect to its parameters (weights $W^{(l)}$ and biases $\mathbf{b}^{(l)}$) are computed by propagating the error signal backward from the subsequent layer $l+1$. The core of this process is to determine $\delta^{(l)}_{\text{batch}} = \frac{\partial C}{\partial \mathbf{Z}^{(l)}_{\text{batch}}}$, which is the gradient of the average batch cost C with respect to the pre-activations $\mathbf{Z}^{(l)}_{\text{batch}}$ of layer l. Once $\delta^{(l)}_{\text{batch}}$ is known, the gradients for $W^{(l)}$ and $\mathbf{b}^{(l)}$ can be found using a similar mechanism as for the output layer.

The derivation proceeds in two main steps for a generic hidden layer l:

1. **Error Propagation (Subsection 5.2.1):** We derive the formula to compute $\delta^{(l)}_{\text{batch}}$ using the error signal from the next layer, $\delta^{(l+1)}_{\text{batch}}$, the weights $W^{(l+1)}$ connecting layer l to $l+1$, and the derivative of the activation function of layer l.
2. **Gradients for Hidden Layer Parameters $W^{(l)}$ and $\mathbf{b}^{(l)}$ (Subsection 5.2.2):** We present the formulas to compute $\frac{\partial C}{\partial W^{(l)}}$ and $\frac{\partial C}{\partial \mathbf{b}^{(l)}}$ using $\delta^{(l)}_{\text{batch}}$ and the activations from the previous layer $\mathbf{H}^{(L-1)}_{\text{batch}}$.

As before, we start with derivations for a single data point x and then generalize to a mini-batch of n samples.

5.2.1 Error Propagation to Hidden Layer l

To determine how the cost C_x for a single sample x changes with respect to the pre-activations $\mathbf{Z}^{(l)}_x$ in hidden layer l, we need to propagate the error signal $\delta^{(l+1)}_x = \frac{\partial C_x}{\partial \mathbf{Z}^{(l+1)}_x}$ backward from layer $l+1$.

Using the chain rule in matrix/vector form:

$$\frac{\partial C_x}{\partial \mathbf{Z}^{(l)}_x} = \frac{\partial \mathbf{H}^{(l)}_x}{\partial \mathbf{Z}^{(l)}_x} \cdot \frac{\partial \mathbf{Z}^{(l+1)}_x}{\partial \mathbf{H}^{(l)}_x} \cdot \frac{\partial C_x}{\partial \mathbf{Z}^{(l+1)}_x}$$

Let's analyze each Jacobian matrix term:

- $\frac{\partial C_x}{\partial \mathbf{Z}^{(l+1)}_x}$ is the error signal from layer $l+1$, which is precisely $\delta^{(l+1)}_x$ (an $n_{l+1} \times 1$ vector).
- $\frac{\partial \mathbf{Z}^{(l+1)}_x}{\partial \mathbf{H}^{(l)}_x}$: Since $\mathbf{Z}^{(l+1)}_x = W^{(l+1)}\mathbf{H}^{(l)}_x + \mathbf{b}^{(l+1)}$, the Jacobian of $\mathbf{Z}^{(l+1)}_x$ with respect to $\mathbf{H}^{(l)}_x$ is the weight matrix $W^{(l+1)}$ (dimensions $n_{l+1} \times n_l$).
- $\frac{\partial \mathbf{H}^{(l)}_x}{\partial \mathbf{Z}^{(l)}_x}$: Since the activation function $g^{(l)}$ is applied element-wise ($H^{(l)}_{x,j} = g^{(l)}(Z^{(l)}_{x,j})$), this Jacobian is a diagonal matrix where the diagonal elements are $g'^{(l)}(Z^{(l)}_{x,j})$. When multiplying with a vector, this is equivalent to an element-wise product with the vector $g'^{(l)}(\mathbf{Z}^{(l)}_x)$.

The gradient $\delta^{(l)}_x = \frac{\partial C_x}{\partial \mathbf{Z}^{(l)}_x}$ is obtained by:

1. Taking the error from the next layer, $\delta^{(l+1)}_x$.

2. Propagating it backward through the weights $W^{(l+1)}$. This involves pre-multiplying by the transpose of the weight matrix: $(W^{(l+1)})^T \delta_x^{(l+1)}$. The resulting vector has dimensions $n_l \times 1$. Each component j of this vector, $\left((W^{(l+1)})^T \delta_x^{(l+1)}\right)_j = \sum_{k=1}^{n_{l+1}} W_{kj}^{(l+1)} (\delta_x^{(l+1)})_k$, represents $\frac{\partial C_x}{\partial H_{x,j}^{(l)}}$, i.e., the sum of error contributions from layer $l+1$ to the activation $H_{x,j}^{(l)}$.
3. Multiplying this propagated error element-wise by the derivative of layer l's activation function, $g'^{(l)}(\mathbf{Z}_x^{(l)})$.

Thus, for a single data point x, the error signal $\delta_x^{(l)}$ is:

$$\delta_x^{(l)} = \left((W^{(l+1)})^T \delta_x^{(l+1)}\right) \odot g'^{(l)}(\mathbf{Z}_x^{(l)}) \tag{13}$$

where $\odot$ denotes the element-wise product. The term $g'^{(l)}(\mathbf{Z}_x^{(l)})$ is an $n_l \times 1$ vector where the j-th element is $g'^{(l)}(Z_{x,j}^{(l)})$.

Extension to Mini-Batch For a mini-batch of n samples, where $C = \frac{1}{n}\sum_{i=1}^{n} C_{x_i}$ is the average cost, we compute $\delta_{\text{batch}}^{(l)} = \frac{\partial C}{\partial \mathbf{Z}_{\text{batch}}^{(l)}}$. The error signal from layer $l+1$ is $\delta_{\text{batch}}^{(l+1)}$ (an $n_{l+1} \times n$ matrix), which, by our convention, already incorporates the $1/n$ averaging factor. The pre-activations at layer l for the batch form the matrix $\mathbf{Z}_{\text{batch}}^{(l)}$ (an $n_l \times n$ matrix). The propagation rule (Eq. (13)) applies column-wise to the matrices:

$$\delta_{\text{batch}}^{(l)} = \left((W^{(l+1)})^T \delta_{\text{batch}}^{(l+1)}\right) \odot g'^{(l)}(\mathbf{Z}_{\text{batch}}^{(l)}) \tag{14}$$

Here, $(W^{(l+1)})^T$ (dimensions $n_l \times n_{l+1}$) multiplies $\delta_{\text{batch}}^{(l+1)}$ (dimensions $n_{l+1} \times n$) to produce an $n_l \times n$ matrix. The term $g'^{(l)}(\mathbf{Z}_{\text{batch}}^{(l)})$ is also an $n_l \times n$ matrix obtained by applying the derivative $g'^{(l)}$ element-wise. The $\odot$ is then an element-wise product between these two $n_l \times n$ matrices.

5.2.2 Gradients for Hidden Layer Parameters $(W^{(l)}, \mathbf{b}^{(l)})$ from $\delta_{\text{batch}}^{(l)}$

Once $\delta_{\text{batch}}^{(l)} = \frac{\partial C}{\partial \mathbf{Z}_{\text{batch}}^{(l)}}$ is known, we can compute the gradients for the weights $W^{(l)}$ and biases $\mathbf{b}^{(l)}$ of hidden layer l. Let $\mathbf{H}_{\text{batch}}^{(l-1)}$ be the $n_{l-1} \times n$ matrix of activations from the previous layer $l-1$. The relationship $Z_{j,i}^{(l)} = \sum_{k=1}^{n_{l-1}} W_{jk}^{(l)} H_{k,i}^{(l-1)} + b_j^{(l)}$ still holds.

Gradient w.r.t. Weights $W^{(l)}$ The derivation for $\frac{\partial C}{\partial W_{pq}^{(l)}}$ is analogous to that for the output layer parameters (Eq. (11)), replacing L with l:

$$\frac{\partial C}{\partial W_{pq}^{(l)}} = \sum_{i=1}^{n} \frac{\partial C}{\partial Z_{p,i}^{(l)}} \frac{\partial Z_{p,i}^{(l)}}{\partial W_{pq}^{(l)}} = \sum_{i=1}^{n} (\delta_{\text{batch}}^{(l)})_{p,i} H_{q,i}^{(l-1)}$$

In matrix form:

$$\frac{\partial C}{\partial W^{(l)}} = \delta_{\text{batch}}^{(l)} (\mathbf{H}_{\text{batch}}^{(l-1)})^T \tag{15}$$

Gradient w.r.t. Biases $\mathbf{b}^{(l)}$ Similarly, for the bias $b_p^{(l)}$ of the p-th neuron in hidden layer l, analogous to Eq. (12):

$$\frac{\partial C}{\partial b_p^{(l)}} = \sum_{i=1}^{n} \frac{\partial C}{\partial Z_{p,i}^{(l)}} \frac{\partial Z_{p,i}^{(l)}}{\partial b_p^{(l)}} = \sum_{i=1}^{n} (\delta_{\text{batch}}^{(l)})_{p,i} \cdot 1$$

For the entire bias vector $\mathbf{b}^{(l)}$ (an $n_l \times 1$ vector):

$$\left(\frac{\partial C}{\partial \mathbf{b}^{(l)}} \right)_p = \sum_{i=1}^{n} (\delta_{\text{batch}}^{(l)})_{p,i} \quad \text{for each } p \in \{1, \ldots, n_l\} \tag{16}$$

As with the output layer, since $\delta_{\text{batch}}^{(l)}$ is defined as the gradient of the *average* batch cost C, it already incorporates the $\frac{1}{n}$ scaling factor.

5.3 Code

```
def _back_propagation(self, y_batch):
        """
        Note: All derivations can be found in the report.
        Computes the gradient of the loss function with respect to all
        parameters using the backpropagation algorithm.
        Stores computed gradients in self.dW and self.db.
        Input parameters:
            - y_batch: True labels for the current batch.
        """

        self.dW = [np.zeros_like(w) for w in self.weights]
        self.db = [np.zeros_like(b) for b in self.biases]
        m = y_batch.shape[0]

        # Step 1: Compute gradients for the output layer
        if self.task == 'classification':
            # For Softmax + Cross-Entropy loss
            y_one_hot = np.zeros_like(self.layers[-1])
            y_one_hot[np.arange(m), y_batch] = 1
            dZ = 1/m * (self.layers[-1] - y_one_hot)

        elif self.task == 'regression':
            # For Linear output + Mean Squared Error loss:
            y_true_reshaped = y_batch.reshape(-1, 1)
            dZ = (2 / m) * (self.layers[-1] - y_true_reshaped)

        if self.num_hidden_layers > 0:
            A_prev = self.layers[-2]
        else:
            A_prev = self.X_batch_current

        self.dW[-1] = np.dot(dZ.T, A_prev)
        self.db[-1] = np.sum(dZ, axis=0)

        # Step 2: Propagate gradients backwards through hidden layers
        for layer in range(self.num_layers - 2, -1, -1):

            W_next = self.weights[layer + 1]
```

```
            dZ_next = dZ

            dA_curr = np.dot(dZ_next, W_next)

            Z_curr = self.cache_Z[layer]
            activation_fn_curr = self.activation_functions[layer]

            if activation_fn_curr == 'ReLU':
                d_activation = np.where(Z_curr > 0, 1, 0)
            else:
                raise ValueError(f"Derivative for activation function {
    activation_fn_curr} not implemented.")

            dZ = dA_curr * d_activation

            if layer == 0:
                A_prev_curr = self.X_batch_current
            else:
                A_prev_curr = self.layers[layer - 1]

            self.dW[layer] = np.dot(dZ.T, A_prev_curr)
            self.db[layer] = np.sum(dZ, axis=0)
```

Listing 1: Custom Backpropagation Algorithm

6 Numerical Gradient Checking

The derivation and implementation of the backpropagation algorithm involve some tough calculus and matrix operations, making it susceptible to errors. To verify the correctness of our analytically derived gradients and their translation into the `_back_propagation` method, we employed numerical gradient checking. This technique provides an independent means of approximating gradients, which can then be compared against those computed by our algorithm. Our numerical gradient checking section comes from the official notes of the Stanford University course on Deep Learning for Computer Vision (CS231n) [7].

Principle of Numerical Gradient Approximation The core idea of numerical gradient checking is to approximate the partial derivative of the cost function C with respect to a specific parameter θ_p (e.g., a single weight $W_{jk}^{(l)}$ or bias $b_j^{(l)}$) using the finite difference method. The two-sided formula, chosen for its higher accuracy, is given by:

$$\frac{\partial C}{\partial \theta_p} \approx \frac{C(\theta_1, \dots, \theta_p + \epsilon, \dots) - C(\theta_1, \dots, \theta_p - \epsilon, \dots)}{2\epsilon} \tag{17}$$

where ϵ is a small perturbation (e.g., 10^{-5}). The terms $C(\dots, \theta_p + \epsilon, \dots)$ and $C(\dots, \theta_p - \epsilon, \dots)$ represent the cost computed by performing a full forward pass through the network with the parameter θ_p perturbed by $+\epsilon$ and $-\epsilon$ respectively, while all other parameters remain unchanged.

Procedure in Our Implementation Our `_gradient_check` method automates this process:

1. **Obtain Analytical Gradients:** First, a forward pass and a backward pass are performed on a small, fixed batch of input data ($\mathbf{X}_{\text{check}}, \mathbf{Y}_{\text{check}}$) using the `_forward_propagation` and `_back_propagation` methods. This yields the analytical gradients $\frac{\partial C}{\partial W^{(l)}}$ and $\frac{\partial C}{\partial \mathbf{b}^{(l)}}$ for all parameters.
2. **Compute Numerical Gradients:** For each parameter θ_p (iterating through every weight and bias in the network):
   (a) The parameter θ_p is temporarily increased by ϵ. A full forward pass is executed with these perturbed parameters, and the cost C^+ is computed using the standard loss function (cross-entropy or MSE).
   (b) The parameter θ_p is then temporarily decreased by ϵ from its original value. Another full forward pass and cost computation yield C^-.
   (c) The numerical gradient for θ_p is calculated using Equation (17).
   (d) The parameter θ_p is restored to its original value before proceeding to the next parameter.
3. **Compare Gradients:** The vector of all numerically computed gradients ($\mathbf{g}_{\text{num}}$) is compared against the vector of all analytically computed gradients ($\mathbf{g}_{\text{ana}}$). The comparison uses the relative error:

$$\text{Relative Error} = \frac{\|\mathbf{g}_{\text{num}} - \mathbf{g}_{\text{ana}}\|_2}{\|\mathbf{g}_{\text{num}}\|_2 + \|\mathbf{g}_{\text{ana}}\|_2 + \eta} \tag{18}$$

   where $\|\cdot\|_2$ is the Euclidean norm, and η is a small constant (e.g., 10^{-12}) to prevent division by zero if both gradient norms are close to zero.

It is important to note that numerical gradient checking is computationally expensive due to the necessity of performing two forward passes for every single parameter in the network. Thus, it is typically used as a debugging tool on small network configurations and small data batches during development and then disabled for actual training.

Verification Outcome The numerical gradient checking procedure was implemented and executed for our custom MLP on both classification and regression configurations using a small dataset. A value of $\epsilon = 10^{-5}$ was used for the perturbation. The relative error (Equation (18)) between the analytically computed gradients from our `_back_propagation` method and the numerically approximated gradients was consistently found to be below 10^{-7}, ensuring that the gradients computed in the backpropagation implementation are correct.

6.1 Code

```
1
2     def gradient_check(self, X, y, epsilon=1e-5, tolerance=1e-7):
3         """
4         Performs numerical gradient checking for ALL weights and biases.
5         Compares backpropagation gradients with numerical approximation.
6         """
7
8         # 0. Store original parameters to restore them later and prevent state changes
```

```
        original_weights = [w.copy() for w in self.weights]
        original_biases = [b.copy() for b in self.biases]

        # 1. Compute analytical gradients using current parameters
        self._forward_propagation(X)
        self._back_propagation(y)

        analytical_grads_flat = []
        for dW_layer in self.dW:
            analytical_grads_flat.extend(dW_layer.flatten().tolist())
        for db_layer in self.db:
            analytical_grads_flat.extend(db_layer.flatten().tolist())

        numerical_grads_flat = []

        print("Checking Weights...")
        # Iterate through each weight matrix
        for l_idx in range(len(self.weights)):
            W_layer_ref = self.weights[l_idx]
            print(f"  Layer {l_idx} weights ({W_layer_ref.shape})...")
            # Iterate through each element of the weight matrix
            for i in range(W_layer_ref.shape[0]):
                for j in range(W_layer_ref.shape[1]):
                    original_val = W_layer_ref[i, j]

                    # Calculate loss for W_ij + epsilon
                    W_layer_ref[i, j] = original_val + epsilon
                    self._forward_propagation(X)
                    loss_plus = self._compute_loss(y, batch_n=1, return_only=True)

                    # Calculate loss for W_ij - epsilon
                    W_layer_ref[i, j] = original_val - epsilon
                    self._forward_propagation(X)
                    loss_minus = self._compute_loss(y, batch_n=1, return_only=True)

                    numerical_grad = (loss_plus - loss_minus) / (2 * epsilon)
                    numerical_grads_flat.append(numerical_grad)

                    # Restore original value for this specific weight
                    W_layer_ref[i, j] = original_val
        print("Checking Biases...")
        # Iterate through each bias vector
        for l_idx in range(len(self.biases)):
            b_layer_ref = self.biases[l_idx] # This is a reference
            print(f"  Layer {l_idx} biases ({b_layer_ref.shape})...")
            # Iterate through each element of the bias vector
            for i in range(b_layer_ref.shape[0]):
                original_val = b_layer_ref[i]

                # Calculate loss for b_i + epsilon
                b_layer_ref[i] = original_val + epsilon
                self._forward_propagation(X)
                loss_plus = self._compute_loss(y, batch_n=1, return_only=True)

                # Calculate loss for b_i - epsilon
                b_layer_ref[i] = original_val - epsilon
                self._forward_propagation(X)
```

```
                loss_minus = self._compute_loss(y, batch_n=1, return_only=True)

                numerical_grad = (loss_plus - loss_minus) / (2 * epsilon)
                numerical_grads_flat.append(numerical_grad)

                # Restore original value for this specific bias
                b_layer_ref[i] = original_val

        # 3. Restore all original parameters from copies to ensure MLP state is unchanged
        self.weights = [w.copy() for w in original_weights]
        self.biases = [b.copy() for b in original_biases]

        # 4. Compute and compare relative errors
        analytical_grads_flat = np.array(analytical_grads_flat)
        numerical_grads_flat = np.array(numerical_grads_flat)

        # Ensure dimensions match before comparison (debugging step)
        if analytical_grads_flat.shape != numerical_grads_flat.shape:
            print(f"WARNING: Shape mismatch! Analytical: {analytical_grads_flat.shape}, Numerical: {numerical_grads_flat.shape}")

        numerator = np.linalg.norm(analytical_grads_flat - numerical_grads_flat)
        denominator = np.linalg.norm(analytical_grads_flat) + np.linalg.norm(numerical_grads_flat)

        relative_diff = numerator / (denominator + 1e-12) # Add epsilon to avoid division by zero

        print(f"Overall gradient relative error: {relative_diff:.2e}")

        if relative_diff >= tolerance:
            raise AssertionError(f"Gradient check failed: Relative error {relative_diff:.2e} >= tolerance {tolerance:.2e}")

        print("Gradient check passed.")
        return relative_diff
```

Listing 2: Custom Numerical Gradient Checking method

7 Conclusion

In this project, we explored the backpropagation algorithm. We covered its theoretical derivation, analyzed its complexity, implemented it from scratch, and verified the implementation by numerical gradient checking.

We began by deriving the general backpropagation equations for a feed-forward neural network, highlighting its reliance on the chain rule for efficient gradient computation.

Then, we analyzed its complexity, noticing that both the forward and backward passes scale with the number of layers and the number of neurons per layer. Additionally, we examined the space complexity and found it also scales linearly with the batch size due to the need to store intermediate activations and gradients during training. These insights reinforce why backpropagation remains tractable even in deep networks: its computational and memory costs grow predictably with model size and batch size. Its efficiency, however, goes beyond just the raw operational count; it is deeply rooted in *how* the chain rule is applied. Firstly, backpropagation can be implemented using a dynamic programming approach. By computing gradients layer-by-layer backwards, intermediate derivatives are calculated once and crucially reused to update all parameters within that layer and propagate the error to the previous layer. This avoids the massive re-computation inherent in naive gradient calculations. Secondly, the derived update rules exhibit locality: gradients for layer l depend only on $\delta^{(l)}$ and the previous layer's activations $h^{(l-1)}$. This modularity, arising from the structured decomposition via the chain rule, simplifies computation. Finally, these mathematical formulations naturally lend themselves to vectorization, mapping directly to efficient matrix operations as implemented. This is critical for practical performance and benefits significantly from hardware-accelerated matrix operations and parallelization strategies used in modern libraries.

This theoretical foundation was then translated into a from-scratch MLP implemented in Python, only using NumPy. The custom MLP is capable of addressing both K-class classification and unidimensional regression tasks. Detailed derivations for the gradients specific to these tasks, including softmax with cross-entropy loss for classification and linear output with mean squared error for regression, were meticulously developed and integrated into the `_back_propagation` method of our MLP.

A critical component of this work was the verification of our analytical gradient calculations. The numerical gradient checking procedure, employing the finite difference method, confirmed the correctness of our backpropagation implementation. The relative error between the analytical gradients and their numerical approximations was consistently found to be below the 10^{-7} threshold, instilling high confidence in the accuracy of our algorithm.

Through this project, several key insights were reinforced. The backpropagation algorithm is essential for the efficient training of neural networks, transforming a potentially intractable gradient computation problem into a manageable one. The process also highlighted the sensitivity of neural network performance to architectural choices and hyperparameter settings, such as learning rate, batch size, and network depth/width, which, while not the primary focus of this study, are critical for achieving optimal results in practice.

Future work could extend this custom MLP framework in several directions. This includes incorporating a wider range of activation functions (e.g., tanh, leaky ReLU) and their derivatives into the backpropagation logic. Expanding the regression capabilities to handle multi-dimensional outputs would also be a valuable addition. Furthermore, implementing regularization techniques such as L1/L2 regularization or dropout and deriving the corresponding modifications to the backpropagation algorithm would enhance the model's ability to generalize and prevent overfitting. These extensions would further solidify the practical utility and educational value of the developed MLP class.

References

[1] F. Rosenblatt, "The perceptron: A probabilistic model for information storage and organization in the brain.," *Psychological review*, vol. 65, no. 6, p. 386, 1958.

[2] K. Hornik, M. Stinchcombe, and H. White, "Universal approximation of an unknown mapping and its derivatives using multilayer feedforward networks," *Neural networks*, vol. 3, no. 5, pp. 551–560, 1990.

[3] D. E. Rumelhart, G. E. Hinton, and R. J. Williams, "Learning representations by back-propagating errors," *nature*, vol. 323, no. 6088, pp. 533–536, 1986.

[4] I. Goodfellow, Y. Bengio, A. Courville, and Y. Bengio, *Deep learning*. MIT press Cambridge, 2016, vol. 1.

[5] C. M. Bishop and N. M. Nasrabadi, *Pattern recognition and machine learning*. Springer, 2006, vol. 4.

[6] K. He, X. Zhang, S. Ren, and J. Sun, "Delving deep into rectifiers: Surpassing human-level performance on imagenet classification," in *Proceedings of the IEEE international conference on computer vision*, 2015, pp. 1026–1034.

[7] Stanford University, *Cs231n: Deep learning for computer vision — gradient checks section*, `https://cs231n.github.io/neural-networks-3/#eval`, Accessed: 2025-05-19, 2025.

[8] K. Jarrett, K. Kavukcuoglu, M. Ranzato, and Y. LeCun, "What is the best multi-stage architecture for object recognition?" In *2009 IEEE 12th international conference on computer vision*, IEEE, 2009, pp. 2146–2153.

[9] V. Nair and G. E. Hinton, "Rectified linear units improve restricted boltzmann machines," in *Proceedings of the 27th international conference on machine learning (ICML-10)*, 2010, pp. 807–814.

[10] X. Glorot, A. Bordes, and Y. Bengio, "Deep sparse rectifier neural networks," in *Proceedings of the fourteenth international conference on artificial intelligence and statistics*, JMLR Workshop and Conference Proceedings, 2011, pp. 315–323.

A Custom MLP implementation

```python
class MLP:
    """
    This class implements a Multi-Layer Perceptron (MLP) neural network.
    It supports both classification and regression tasks.
    The backpropagation algorithm gradients can be checked using numerical gradient checking.
    The MLP can be trained using mini-batch gradient descent.
    """

    ALLOWED_HIDDEN_ACTIVATION_FUNCTIONS = ['ReLU']
    ALLOWED_TASKS = ['classification', 'regression']

    def __init__(self, hidden_layer_neurons: list = [],
                 activation_functions: list = ['ReLU'],
                 task:str = 'classification',verbose:bool = True):
        """
        Builds a Multi Layer Perceptron architecture and initializes its weights.

        Input Parameters:
            - hidden_layer_neurons: Number of nodes for each hidden layer .
            - activation_functions: Set the activation function for each layer.
            - task: task of the MLP.
            - verbose: If True, prints information while interacting with the MLP.
        """

        # Some checks on data provided
        if task not in self.ALLOWED_TASKS: raise ValueError(f"Task should be one of the following: {self.ALLOWED_TASKS}")
        if len(hidden_layer_neurons) < 0: raise ValueError(f"Number of hidden layers should be >= 0")

        # Set up the Multi-Layer Perceptron architecture: Number of layers, neurons per layer, activation functions.
        self.task = task
        self.num_hidden_layers = len(hidden_layer_neurons)
        self.num_layers = len(hidden_layer_neurons) +1
        self.layer_neurons = hidden_layer_neurons
        self.num_neurons = np.sum(self.layer_neurons)
        self.activation_functions = self._set_activation_functions(activation_functions)

        # Initialize the parameters
        self.weights = None
        self.biases = None

        # Set the verbose flag
        self.verbose = verbose

        # Initialize caches and gradient holders
        self.cache_Z = []  # To store pre-activations (Z values) for each layer
        self.X_batch_current = None # To store the X_batch used in the current forward pass
        self.dW = None     # To store weight gradients
        self.db = None     # To store bias gradients


```

```
        if self.verbose:
            print("#################################################################")
            print(f"Initialized the following Multi Layer Perceptron: \n"
                f"    -Number of hidden layers: {self.num_hidden_layers} \n"
                f"    -Number of neurons per layer: {self.layer_neurons} \n"
                f"    -Activation functions: {self.activation_functions} \n"
                f"    -Task: {self.task} \n")

    def train(self, X,y, batch_size:int = 1, epochs:int = 1,learning_rate: float = 0.001,):
        """
        Trains the MLP for the given data, task, architecture and batch size.

        Input parameters:
            - X: Input data of shape (n_samples, n_features).
            - y: Target data of shape (n_samples, n_classes) for classification or (n_samples,) for regression.
            - batch_size (int): The number of data samples to process per batch during training.
            - epochs: Number of times the MLP will see the training data.
            - learning_rate: The learning rate for the training algorithm
        """

        X = np.asarray(X)
        y = np.asarray(y)

        if self.verbose:
            print("#################################################################")
            print("Fitting the MLP for the given data:")

        # Initialize weights, biases, layers given the data structure
        self._fit(X,y, batch_size=batch_size)

        if self.verbose:
            print("#################################################################")
            print("Training the MLP for the given data:")

        # Train the MLP
        self.learning_rate = learning_rate
        n_samples = X.shape[0]

        for epoch in range(epochs):
            if self.verbose: print(f"Epoch {epoch + 1}/{epochs}")

            indices = np.random.permutation(n_samples)
            print("Training epoch ", epoch + 1, "with batch size", batch_size)
            X_shuffled = X[indices]
            y_shuffled = y[indices]

            batch_n = 1
            for start in range(0, n_samples, batch_size):
                end = min(start + batch_size, n_samples)
                X_batch = X_shuffled[start:end]
                y_batch = y_shuffled[start:end]

                # Train within the current branch
                self._forward_propagation(X_batch)
```

```
                self._compute_loss(y_batch, batch_n)
                self._back_propagation(y_batch)
                self._update_weights()
                batch_n += 1

        if self.verbose: print("Finished training")

    def predict(self, X_test):
        """
        Makes predictions on a data set X_test.

        Input parameters:
            - X_test: Input data for which to make predictions of shape (n_samples, n_features).

        Returns:
            - predictions: Predicted values.
        """
        if self.weights is None or self.biases is None:
            raise RuntimeError("The model has not been trained yet. Call train() first.")

        # Ensure X_test is a 2D NumPy array
        X_test = np.asarray(X_test)

        expected_features = self.weights[0].shape[1]

        if X_test.ndim == 1:
            if X_test.shape[0] == expected_features:
                X_test = X_test.reshape(1, -1)
            else:
                raise ValueError(
                    f"Input X_test is 1D with {X_test.shape[0]} elements. "
                    f"If it's a single sample, it should have {expected_features} features. "
                    "Otherwise, X_test should be 2D (samples, features)."
                )
        elif X_test.ndim != 2:
            raise ValueError(f"Input data X_test must be 2-dimensional (samples, features). Got {X_test.ndim} dimensions.")

        # Check if number of features in X_test matches model's input features
        if X_test.shape[1] != expected_features:
            raise ValueError(
                f"Input X_test has {X_test.shape[1]} features, "
                f"but the model was trained/initialized expecting {expected_features} features."
            )

        # Perform forward propagation for prediction
        A_prev = X_test
        for layer_idx in range(self.num_layers):
            W = self.weights[layer_idx]
            b = self.biases[layer_idx]

            Z_current = np.dot(A_prev, W.T) + b
            A_current = self._activation(layer_idx, Z_current)
```

```
157
158             A_prev = A_current
159
160         y_pred_raw = A_prev
161
162         if self.task == 'classification':
163             predictions = np.argmax(y_pred_raw, axis=1)
164         elif self.task == 'regression':
165             predictions = y_pred_raw.squeeze() if y_pred_raw.shape[1] == 1 else
    y_pred_raw
166
167         return predictions
168
169     def _set_activation_functions(self, activation_functions):
170         """
171         Set the activation function of each hidden-layer, only allowing those
    predefined in ALLOWED_HIDDEN_ACTIVATION_FUNCTIONS.
172
173         Input parameters:
174             - activation_functions: list of activation functions. If len == 1, the
    same activation function will be used for
175                                     all layers. Otherwise, each layer will have the
    corresponding activation function.
176         """
177
178         # Check if the activation functions are implemented
179         for act in activation_functions:
180             if act not in self.ALLOWED_HIDDEN_ACTIVATION_FUNCTIONS:
181                 raise ValueError(f"Activation function {act} is not implemented yet
    for hidden-layers. Only 'ReLU' is allowed.")
182
183         # Store the activation functions of the hidden layers
184         if len(activation_functions) == 1: act_func = activation_functions * self.
    num_hidden_layers
185         elif len(activation_functions) == self.num_hidden_layers: act_func =
    activation_functions
186         else: raise ValueError(f"Number of activation functions ({len(
    activation_functions)}) does not match number of layers ({self.num_hidden_layers})
    ")
187
188         # Include the output activation function
189         if self.task == 'classification': act_func.append('softmax')
190         elif self.task == 'regression': act_func.append('linear')
191
192         return act_func
193
194     def _fit(self, X, y, batch_size):
195         """
196         Initializes the weight matrices and bias vectors according to the provided
    data and task.
197
198         Input Parameters:
199             - X : training data features
200             - y: training target
201             - batch_size: size of the mini-batch for training
202         """
203         # Some checks on the data:
```

```
204         X = np.asarray(X)
205         if X.ndim < 2: raise ValueError(f"Input data X must be 2-dimensional (samples, features). Got {X.ndim} dimensions.")
206         y = np.asarray(y)
207         if X.shape[0] != y.shape[0]: raise ValueError(f"X and y must have the same number of samples")
208 
209         # Initialize the weights and biases
210         self._init_weights(input_features = X.shape[1], target = y)
211 
212         # Initialize the layers
213         self._init_layers(batch_size, target = y)
214 
215         if self.verbose:
216             print("######################################################################")
217             print("Initial weights and biases: ")
218             print(f"Weights: {self.weights}")
219             print(f"Biases: {self.biases}")
220             print(f"Layers: {self.layers}")
221 
222     def _init_weights(self, input_features, target):
223         """
224         Initialize the weights (He initialization) of the Multi Layer Perceptron for all hidden layers.
225 
226         Input parameters:
227             - input_features: Dimensionality(number of features) of the training data.
228             - target: target variable containing the target for all training samples.
229         """
230 
231         # Initialize the hidden-layer weights and biases
232         weights = []
233         biases = []
234         for layer in range(self.num_hidden_layers):
235             if layer == 0: weight_matrix = np.random.randn(self.layer_neurons[layer], input_features) * np.sqrt(2 / input_features)
236             else: weight_matrix = np.random.randn(self.layer_neurons[layer],self.layer_neurons[layer-1]) * np.sqrt(2 /self.layer_neurons[layer-1])
237             bias_vector = np.zeros(self.layer_neurons[layer])
238             weights.append(weight_matrix)
239             biases.append(bias_vector)
240 
241         # Initialize the output layer weights and biases
242         self.output_neurons = len(np.unique(target)) if self.task == 'classification' else 1
243         if self.num_hidden_layers == 0:
244             num_neurons_in_prev_layer = input_features
245         else:
246             num_neurons_in_prev_layer = self.layer_neurons[-1]
247 
248         output_layer_weights = np.random.randn(self.output_neurons, num_neurons_in_prev_layer) * np.sqrt(2 / num_neurons_in_prev_layer)
249         weights.append(output_layer_weights)
250         biases.append(np.zeros(self.output_neurons))
251 
252         # Save the weights and biases
253         self.weights = weights
```

```
        self.biases = biases

    def _init_layers(self, batch_size, target):
        """
        Initializes layer values to 0.

        Input parameters:
            - batch_size: size of the mini-batch for training
            - target: target variable containing the target for all training samples.
        """

        layers = []
        # Initialize Hidden Layers
        for layer in range(self.num_hidden_layers):
            layers.append(np.zeros(shape=(batch_size,self.layer_neurons[layer])))

        # Initialize Output Layer
        if self.task == 'classification': layers.append(np.zeros(shape=(batch_size,len
(np.unique(target)))))
        elif self.task == 'regression': layers.append(np.zeros(shape=(batch_size,1)))
        self.layers = layers

    def _forward_propagation(self, X):
        """
        Performs a forward step through the network.

        Input parameters:
            - X: Input data of shape (n_samples, n_features).
        """
        self.X_batch_current = X
        self.cache_Z = []

        A = X

        for layer in range(self.num_layers): # 0 to num_layers-1
            W = self.weights[layer]
            b = self.biases[layer]

            # Z = A . W^T + b
            Z = np.dot(A, W.T) + b
            self.cache_Z.append(Z)

            self.layers[layer] = self._activation(layer, Z)
            A = self.layers[layer]

    def _activation(self, layer, Z):
        """
        Computes and returns the activation layer.

        Input parameters:
            - layer: Index of the current layer.
            - Z: Pre-activation values for the current layer.
        """
        if self.activation_functions[layer] == 'ReLU': return np.maximum(0, Z)
        elif self.activation_functions[layer] == 'linear': return Z
        elif self.activation_functions[layer] == 'softmax':
            # We substract max(z) to all elements to avoid big exponentials.
```

```
            # Softmax is shift invariant, so no problem with that.
            e_z = np.exp(Z - np.max(Z, axis=1, keepdims=True))
            return e_z / np.sum(e_z, axis=1, keepdims=True)

    def _compute_loss(self, y_true_batch, batch_n, return_only=False):
        """
        Computes the loss function.

        Input parameters:
            - y_true_batch: True labels for the current batch.
            - batch_n: Index of the current batch.
            - return_only: If True, only returns the loss value without printing it.

        Returns:
            - loss: Computed loss value.
        """
        y_pred = self.layers[-1]

        if self.task == 'classification':
            epsilon = 1e-12
            num_samples = y_true_batch.shape[0]
            correct_class_probabilities = y_pred[np.arange(num_samples), y_true_batch.
    astype(int)]
            log_likelihood = np.log(correct_class_probabilities + epsilon)
            loss = -np.mean(log_likelihood)
        elif self.task == 'regression':
            y_true_reshaped = y_true_batch.reshape(-1, 1)
            loss = np.mean((y_pred - y_true_reshaped)**2)
        else:
            raise ValueError(f"Unknown task: {self.task}")

        if not return_only:
            if self.verbose: print(f"Loss for batch {batch_n} = {loss}")
            self.loss = loss

        return loss

    def _back_propagation(self, y_batch):
        """
        Back-propagation step to compute gradients of the loss function.
        Computes the gradient of the loss function with respect to all
        parameters using the back-propagation algorithm.
        Stores computed gradients in self.dW and self.db.

        Note: All derivations can be found in the report.

        Input parameters:
            - y_batch: True labels for the current batch.
        """

        self.dW = [np.zeros_like(w) for w in self.weights]
        self.db = [np.zeros_like(b) for b in self.biases]
        m = y_batch.shape[0]

        # Step 1: Compute gradients for the output layer
        if self.task == 'classification':
            # For Softmax + Cross-Entropy loss
```

```
            y_one_hot = np.zeros_like(self.layers[-1])
            y_one_hot[np.arange(m), y_batch] = 1
            dZ = 1/m * (self.layers[-1] - y_one_hot)

        elif self.task == 'regression':
            # For Linear output + Mean Squared Error loss:
            y_true_reshaped = y_batch.reshape(-1, 1)
            dZ = (2 / m) * (self.layers[-1] - y_true_reshaped)

        if self.num_hidden_layers > 0:
            A_prev = self.layers[-2]
        else:
            A_prev = self.X_batch_current

        self.dW[-1] = np.dot(dZ.T, A_prev)
        self.db[-1] = np.sum(dZ, axis=0)

        # Step 2: Propagate gradients backwards through hidden layers
        for layer in range(self.num_layers - 2, -1, -1):

            W_next = self.weights[layer + 1]
            dZ_next = dZ

            dA_curr = np.dot(dZ_next, W_next)

            Z_curr = self.cache_Z[layer]
            activation_fn_curr = self.activation_functions[layer]

            if activation_fn_curr == 'ReLU':
                d_activation = np.where(Z_curr > 0, 1, 0)
            else:
                raise ValueError(f"Derivative for activation function {
    activation_fn_curr} not implemented.")

            dZ = dA_curr * d_activation

            if layer == 0:
                A_prev_curr = self.X_batch_current
            else:
                A_prev_curr = self.layers[layer - 1]

            self.dW[layer] = np.dot(dZ.T, A_prev_curr)
            self.db[layer] = np.sum(dZ, axis=0)

    def _update_weights(self):
        """
        Uses Gradient Descent to update the weights and biases of the MLP.
        Assumes self.learning_rate is set and self.dW, self.db contain gradients.
        """

        for i in range(self.num_layers):
            self.weights[i] -= self.learning_rate * self.dW[i]
            self.biases[i] -= self.learning_rate * self.db[i]

    def gradient_check(self, X, y, epsilon=1e-5, tolerance=1e-7):
        """
        Performs numerical gradient checking for ALL weights and biases.
```

```
        Compares backpropagation gradients with numerical approximation.

        Input parameters:
            - X: Input data of shape (n_samples, n_features).
            - y: Target data of shape (n_samples, n_classes) for classification or (n_samples,) for regression.
            - epsilon: Small value for numerical gradient approximation.
            - tolerance: Tolerance for gradient check failure.
        Returns:
            - relative_diff: Relative difference between analytical and numerical gradients.
        """

        # 0. Store original parameters to restore them later and prevent state changes
        original_weights = [w.copy() for w in self.weights]
        original_biases = [b.copy() for b in self.biases]

        # 1. Compute analytical gradients using current parameters
        self._forward_propagation(X)
        self._back_propagation(y)

        analytical_grads_flat = []
        for dW_layer in self.dW:
            analytical_grads_flat.extend(dW_layer.flatten().tolist())
        for db_layer in self.db:
            analytical_grads_flat.extend(db_layer.flatten().tolist())

        numerical_grads_flat = []

        print("Checking Weights...")
        # Iterate through each weight matrix
        for l_idx in range(len(self.weights)):
            W_layer_ref = self.weights[l_idx]
            print(f"  Layer {l_idx} weights ({W_layer_ref.shape})...")
            # Iterate through each element of the weight matrix
            for i in range(W_layer_ref.shape[0]):
                for j in range(W_layer_ref.shape[1]):
                    original_val = W_layer_ref[i, j]

                    # Calculate loss for W_ij + epsilon
                    W_layer_ref[i, j] = original_val + epsilon
                    self._forward_propagation(X)
                    loss_plus = self._compute_loss(y, batch_n=1, return_only=True)

                    # Calculate loss for W_ij - epsilon
                    W_layer_ref[i, j] = original_val - epsilon
                    self._forward_propagation(X)
                    loss_minus = self._compute_loss(y, batch_n=1, return_only=True)

                    numerical_grad = (loss_plus - loss_minus) / (2 * epsilon)
                    numerical_grads_flat.append(numerical_grad)

                    # Restore original value for this specific weight
                    W_layer_ref[i, j] = original_val
        print("Checking Biases...")
        # Iterate through each bias vector
        for l_idx in range(len(self.biases)):
```

```
            b_layer_ref = self.biases[l_idx] # This is a reference
            print(f"  Layer {l_idx} biases ({b_layer_ref.shape})...")
            # Iterate through each element of the bias vector
            for i in range(b_layer_ref.shape[0]):
                original_val = b_layer_ref[i]

                # Calculate loss for b_i + epsilon
                b_layer_ref[i] = original_val + epsilon
                self._forward_propagation(X)
                loss_plus = self._compute_loss(y, batch_n=1, return_only=True)

                # Calculate loss for b_i - epsilon
                b_layer_ref[i] = original_val - epsilon
                self._forward_propagation(X)
                loss_minus = self._compute_loss(y, batch_n=1, return_only=True)

                numerical_grad = (loss_plus - loss_minus) / (2 * epsilon)
                numerical_grads_flat.append(numerical_grad)

                # Restore original value for this specific bias
                b_layer_ref[i] = original_val

        # 3. Restore all original parameters from copies to ensure MLP state is unchanged
        self.weights = [w.copy() for w in original_weights]
        self.biases = [b.copy() for b in original_biases]

        # 4. Compute and compare relative errors
        analytical_grads_flat = np.array(analytical_grads_flat)
        numerical_grads_flat = np.array(numerical_grads_flat)

        # Ensure dimensions match before comparison (debugging step)
        if analytical_grads_flat.shape != numerical_grads_flat.shape:
            print(f"WARNING: Shape mismatch! Analytical: {analytical_grads_flat.shape}, Numerical: {numerical_grads_flat.shape}")

        numerator = np.linalg.norm(analytical_grads_flat - numerical_grads_flat)
        denominator = np.linalg.norm(analytical_grads_flat) + np.linalg.norm(numerical_grads_flat)

        relative_diff = numerator / (denominator + 1e-12) # Add epsilon to avoid division by zero

        print(f"Overall gradient relative error: {relative_diff:.2e}")

        if relative_diff >= tolerance:
            raise AssertionError(f"Gradient check failed: Relative error {relative_diff:.2e} >= tolerance {tolerance:.2e}")

        print("Gradient check passed.")
        return relative_diff
```

Listing 3: Custom Multi Layer Perceptron