



KOSZALIN UNIVERSITY OF TECHNOLOGY

APPLICATIONS OF ARTIFICIAL INTELLIGENCE  
PROJECT REPORT

# Handwritten text symbol recognition with deep neural networks

*Paweł Frankowski Kacper Ochnik*

supervised by  
Dr. Adam Słowik

February 6, 2024

# Contents

<b>1</b>	<b>Introduction</b>	<b>2</b>
<b>2</b>	<b>Our goal</b>	<b>3</b>
2.1	Specific objectives . . . . .	3
2.2	Expected Outcomes . . . . .	3
<b>3</b>	<b>Decision Boundary</b>	<b>4</b>
3.1	Definition . . . . .	4
3.2	Adaptability . . . . .	4
3.3	Optimization Impact . . . . .	4
<b>4</b>	<b>Weights and Biases</b>	<b>5</b>
4.1	Initialization . . . . .	5
4.2	Weighted sum of inputs . . . . .	5
<b>5</b>	<b>Our model</b>	<b>6</b>
<b>6</b>	<b>Activation functions</b>	<b>7</b>
6.1	Sigmoid function . . . . .	7
6.2	The derivative of the sigmoid function . . . . .	7
<b>7</b>	<b>Gradient descent and backpropagation</b>	<b>9</b>
7.1	Gradients of the cost function . . . . .	9
7.2	Data loading and learning . . . . .	10
7.3	Backpropagation . . . . .	11
<b>8</b>	<b>Cost Landscape</b>	<b>12</b>
8.1	Definition . . . . .	12
8.2	Characteristics . . . . .	12
8.3	Impact on Optimization . . . . .	12
8.4	Optimization Strategies . . . . .	12
<b>9</b>	<b>Learning algorithm - naive approach, calculus approach, digit recognition - Kacper</b>	<b>13</b>
<b>10</b>	<b>Chain Rule</b>	<b>14</b>
10.1	Definition . . . . .	14
10.2	Application in Backpropagation . . . . .	14
10.3	Examples from our code . . . . .	14
<b>11</b>	<b>Testing the network</b>	<b>15</b>
<b>12</b>	<b>Conclusion</b>	<b>16</b>

# 1 Introduction

The significance of digit recognition in the realm of artificial intelligence cannot be overstated, particularly in its application to handwritten digits. This project delves into the intricate process of teaching an AI model to discern and interpret handwritten numerals. With a myriad of potential real-world applications, from automating data entry tasks to enhancing accessibility in digital document processing, the ability of an AI model to accurately recognize handwritten digits holds immense promise. Through the utilization of advanced algorithms and machine learning techniques, this project endeavors to unlock the potential of AI in transforming handwritten digit recognition into a seamless and efficient process.

## 2 Our goal

The primary objective of our project is to develop a handwritten text symbol recognition system using deep neural networks. We aimed to create a model capable of accurately identifying and classifying handwritten digits ranging from 0 to 9 on a matrix of 28x28 pixels.

### 2.1 Specific objectives

In pursuit of our overarching goal, we have identified the following specific objectives:

1. **Project setup** - Set up the project environment and install the necessary libraries and packages.
2. **Code implementation** - Write Python code to implement the deep neural network architecture. This includes developing modules for data preprocessing, model training, and evaluation.
3. **Data Collection** - Gather a comprehensive dataset of handwritten digits (0 to 9) in a 28x28 pixel matrix format from MNIST.
4. **Learning** - Train the model using the collected dataset.
5. **Optimization** - Improve the model's accuracy through optimization techniques. Accelerate computational efficiency for faster calculations.
6. **Testing** - Create testing GUI for the trained model. Evaluate the model's performance metrics.
7. **Documentation** - Write a comprehensive report documenting the project's objectives, methodology, and outcomes.

### 2.2 Expected Outcomes

Upon successful completion of our project, we anticipate achieving the following outcomes:

- Develop a robust deep neural network model capable of recognizing and classifying handwritten digits from 0 to 9.
- Train the model to achieve a acceptable level of accuracy.

## 3 Decision Boundary

The concept of a decision boundary is fundamental in understanding how a neural network makes predictions and categorizes inputs. In the context of our handwritten text symbol recognition project, the decision boundary plays a crucial role in distinguishing between different handwritten digits.

### 3.1 Definition

A decision boundary is a hypersurface that separates the input space into regions assigned to different classes. In our case, each region corresponds to a specific digit (0 through 9). The decision boundary is learned during the training phase of the neural network and is a reflection of the network's understanding of the features that differentiate one digit from another.

### 3.2 Adaptability

The adaptability of the decision boundary is a key aspect of the neural network's performance. A well-trained model should be capable of adjusting its decision boundary to accommodate variations in handwriting styles, ensuring robust recognition across different writing patterns.

### 3.3 Optimization Impact

During the optimization process, the neural network adjusts its parameters to optimize the decision boundary for accurate classification. Understanding the decision boundary dynamics sheds light on how changes in weights and biases impact the model's ability to discriminate between different digits.

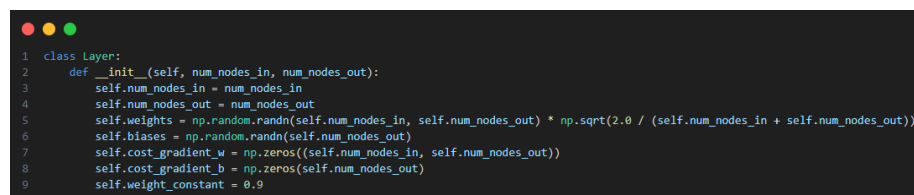
In the subsequent sections, we will delve into the weights and biases optimization process, shedding further light on how these adjustments influence the decision boundary and contribute to the overall performance of our handwritten text symbol recognition system.

## 4 Weights and Biases

Weights are indicators of the importance of the input in predicting the final output. Biases are essential as they allow the network to have some flexibility in fitting the data. In the training process of our neural network, the weights and biases play a crucial role in determining the model's performance. Here, we discuss the initialization, calculation, and adjustment of weights and biases in the network.

### 4.1 Initialization

The `Layer` class represents a single layer in the neural network. It contains information about the number of nodes in and out, weights, biases, and gradients. The neural network is initialized with random weights using the Xavier/Glorot initialization technique. This method helps in achieving better convergence during training.



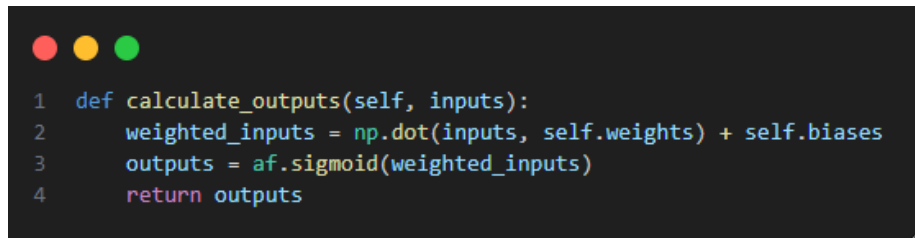
```
1 class Layer:
2     def __init__(self, num_nodes_in, num_nodes_out):
3         self.num_nodes_in = num_nodes_in
4         self.num_nodes_out = num_nodes_out
5         self.weights = np.random.randn(self.num_nodes_in, self.num_nodes_out) * np.sqrt(2.0 / (self.num_nodes_in + self.num_nodes_out))
6         self.biases = np.random.randn(self.num_nodes_out)
7         self.cost_gradient_w = np.zeros((self.num_nodes_in, self.num_nodes_out))
8         self.cost_gradient_b = np.zeros(self.num_nodes_out)
9         self.weight_constant = 0.9
```

Figure 1: Initialization of random biases and weights using Xavier/Glorot technique

When a neural network is first trained, it is first fed with input. Since it isn't trained yet, we don't know which weights should we use for each input. And so, each input is randomly assigned a weight. It will very likely give incorrect output.

### 4.2 Weighted sum of inputs

In the `Layer` class, the calculation of outputs begins with the computation of weighted inputs. For each node in the layer, the weighted inputs are calculated using the formula.:



```
1 def calculate_outputs(self, inputs):
2     weighted_inputs = np.dot(inputs, self.weights) + self.biases
3     outputs = af.sigmoid(weighted_inputs)
4     return outputs
```

Figure 2: The dot product of inputs and weights, representing the weighted sum of inputs

The `calculate_outputs` method calculates the dot product of inputs and weights, representing the weighted sum of inputs. The biases are then added to this sum to introduce flexibility in fitting the data. Finally, the result is passed through the sigmoid activation function to introduce non-linearity to the model, and the outputs are obtained.

## 5 Our model

FFN - Feedforward Neural Network opisac w tym rozdziale opisac jak wyglada nasz model AI. ...

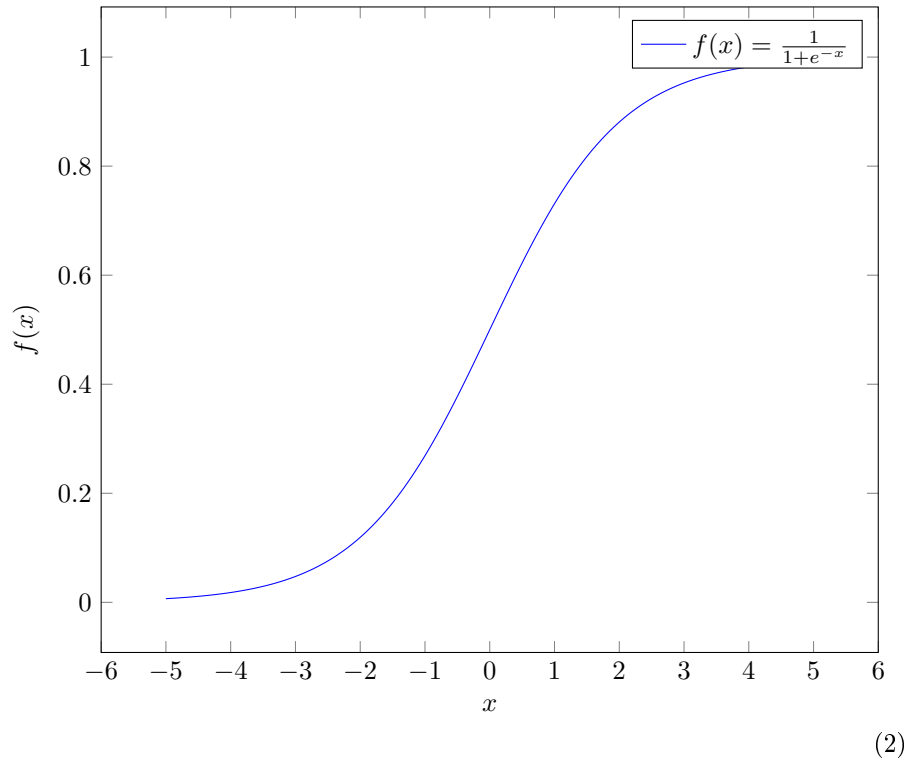
## 6 Activation functions

Activation functions play a crucial role in neural networks, introducing non-linearity to the model and allowing it to learn complex relationships in the data. Here, we describe various activation functions implemented in the `activation functions` module.

### 6.1 Sigmoid function

The sigmoid function squashes its input to the range  $(0, 1)$ , making it suitable for binary classification problems.

$$f(x) = \frac{1}{1 + e^{-x}} \quad (1)$$

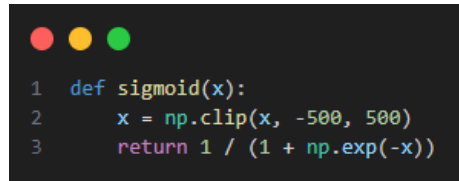


### 6.2 The derivative of the sigmoid function

The derivative of the sigmoid function is used in the backpropagation algorithm to calculate the gradients of the cost function with respect to the weights and biases. The derivative of the sigmoid function is given by the formula:

$$f'(x) = f(x) \cdot (1 - f(x)) \quad (3)$$

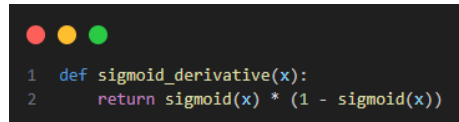


A code editor window with a dark background and three colored window control buttons (red, yellow, green) in the top-left corner. It contains three lines of Python code: a function definition for sigmoid(x), clipping x to the range [-500, 500], and returning the sigmoid value.

```
1 def sigmoid(x):  
2     x = np.clip(x, -500, 500)  
3     return 1 / (1 + np.exp(-x))
```

Figure 3: Sigmoid function

Limiting the values of  $x$  to prevent overflow. Overflow refers to a situation where the exponential function in the sigmoid formula produces very large positive values, causing numerical instability in calculations. The exponential function, especially when dealing with large positive inputs, can lead to floating-point overflow, which means the result becomes too large to be represented within the numerical precision of the system.

A code editor window with a dark background and three colored window control buttons (red, yellow, green) in the top-left corner. It contains two lines of Python code: a function definition for sigmoid\_derivative(x) that returns the product of the sigmoid function and its complement.

```
1 def sigmoid_derivative(x):  
2     return sigmoid(x) * (1 - sigmoid(x))
```

Figure 4: The derivative of the sigmoid function

## 7 Gradient descent and backpropagation

### 7.1 Gradients of the cost function

**Gradient Descent** - Is an optimization algorithm that is used to find the weights that minimize the cost function. We need two things to do so. Direction in which to navigate and the size of the steps for navigating. To know which direction to navigate the cost function, gradient descent uses backpropagation. More specifically, it uses the gradients calculated through backpropagation. These gradients are used for determining the direction to navigate to find the minimum point. Descending slope will lead us to the minimum point. The cost gradients for weights and biases are initialized with zeros, and during training, these gradients will be updated based on the backpropagation algorithm [1]

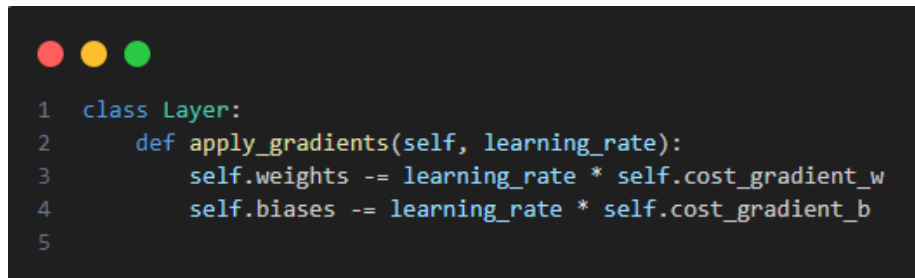
```
1 class Layer:
2     def __init__(self, num_nodes_in, num_nodes_out):
3         self.num_nodes_in = num_nodes_in
4         self.num_nodes_out = num_nodes_out
5         self.weights = np.random.randn(self.num_nodes_in, self.num_nodes_out) * np.sqrt(2.0 / (self.num_nodes_in + self.num_nodes_out))
6         self.biases = np.random.randn(self.num_nodes_out)
7         self.cost_gradient_w = np.zeros((self.num_nodes_in, self.num_nodes_out))
8         self.cost_gradient_b = np.zeros(self.num_nodes_out)
9         self.weight_constant = 0.9
```

Figure 5: Initialization a vector of zeros with the same length as the bias vector

This matrix will be used to accumulate the gradients of the cost function with respect to the weights during the backpropagation process.

```
1 def apply_gradients(self, learning_rate):
2     for layer in self.layers:
3         if self.layers.index(layer) == 0:
4             continue
5         else:
6             layer.apply_gradients(learning_rate)
```

Figure 6: The `apply_gradients` method in `Neural network` class updates the weights and biases for each layer.

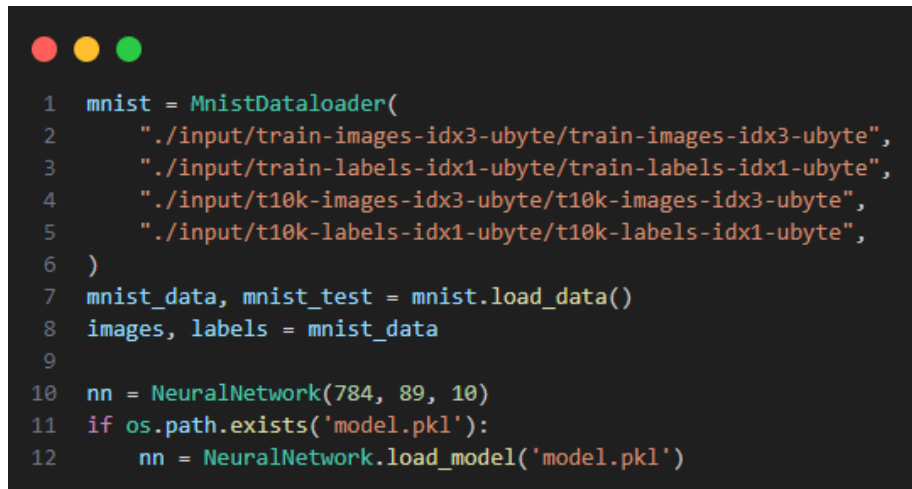


```
1 class Layer:
2     def apply_gradients(self, learning_rate):
3         self.weights -= learning_rate * self.cost_gradient_w
4         self.biases -= learning_rate * self.cost_gradient_b
5
```

Figure 7: The `apply_gradients` method in `Layer` class updates the weights and biases for a layer.

The `apply_gradients` method is responsible for updating the weights and biases of a layer based on the calculated gradients.

## 7.2 Data loading and learning



```
1  mnist = MnistDataLoader(  
2      "./input/train-images-idx3-ubyte/train-images-idx3-ubyte",  
3      "./input/train-labels-idx1-ubyte/train-labels-idx1-ubyte",  
4      "./input/t10k-images-idx3-ubyte/t10k-images-idx3-ubyte",  
5      "./input/t10k-labels-idx1-ubyte/t10k-labels-idx1-ubyte",  
6  )  
7  mnist_data, mnist_test = mnist.load_data()  
8  images, labels = mnist_data  
9  
10 nn = NeuralNetwork(784, 89, 10)  
11 if os.path.exists('model.pkl'):  
12     nn = NeuralNetwork.load_model('model.pkl')
```

Figure 8: Data loading from mnist and creating a neural network

A neural network is instantiated with the specified architecture: 784 input nodes, one hidden layer with 89 nodes, and 10 output nodes. Checks if a pre-trained model file `model.pkl` exists. If it does, the model is loaded from the file. Otherwise, the model is trained using the MNIST dataset.

### 7.3 Backpropagation

**Backpropagation** - is a training algorithm for feedforward neural networks [2] that propagates the error backward from the output to the input layer. It plays a crucial role in gradient descent, a process of finding the minimum of the cost function. Gradient descent relies on backpropagation to calculate gradients by moving backward in the neural network. Together, backpropagation and gradient descent is used for the purpose of improving the prediction accuracy of neural networks. They help improve prediction accuracy by reducing the output error in neural networks.

## 8 Cost Landscape

Understanding the cost landscape is essential for comprehending the optimization process in neural networks. In our project of handwritten text symbol recognition, the cost landscape reflects the relationship between the model's parameters and the cost function, providing insights into the optimization journey.

### 8.1 Definition

The cost landscape, also known as the loss landscape, is a graphical representation of how the cost function varies with changes in the neural network's parameters, such as weights and biases. It visualizes the "landscape" of the optimization problem, showing areas of high and low cost values.

### 8.2 Characteristics

The cost landscape can exhibit various characteristics, including multiple local minima, saddle points, and plateaus. Local minima represent suboptimal solutions, while saddle points pose challenges to gradient-based optimization algorithms. Plateaus, on the other hand, indicate regions of slow convergence, where the optimization process may stagnate.

### 8.3 Impact on Optimization

Navigating the cost landscape effectively is crucial for optimizing the performance of the neural network. Understanding the topology of the landscape helps in selecting appropriate optimization algorithms, setting learning rates, and initializing parameters to avoid getting stuck in local minima or encountering convergence issues.

### 8.4 Optimization Strategies

Based on the insights gained from analyzing the cost landscape, various optimization strategies can be employed to navigate the landscape effectively. Techniques such as stochastic gradient descent, momentum optimization, and adaptive learning rates aim to mitigate the challenges posed by the landscape topology and accelerate convergence towards the global minimum.

In the subsequent sections, we will explore the optimization strategies employed in our project and their impact on improving the performance of our handwritten text symbol recognition system.

## **9 Learning algorithm - naive approach, calculus approach, digit recognition - Kacper**

Kacper you should check other sections because there could be duplicates! ...

## 10 Chain Rule

The chain rule is a fundamental concept in calculus that describes how to compute the derivative of a composite function. In the context of neural networks, the chain rule plays a crucial role in the backpropagation algorithm, allowing us to efficiently calculate the gradients of the loss function with respect to the network's parameters.

### 10.1 Definition

The chain rule states that if  $f$  and  $g$  are differentiable functions, then the derivative of their composition  $f(g(x))$  with respect to  $x$  is given by:

$$(f \circ g)'(x) = f'(g(x)) \cdot g'(x)$$

In the context of neural networks, this means that to compute the gradient of the loss function with respect to the weights and biases of a particular layer, we need to multiply the gradient of the loss function with respect to the output of that layer by the gradient of the output of that layer with respect to its inputs.

### 10.2 Application in Backpropagation

During the backpropagation process, the chain rule is applied recursively to propagate gradients backward through the network. Starting from the output layer and moving backward through each layer, the chain rule is used to compute the gradients of the loss function with respect to the activations of each layer. These gradients are then used to update the weights and biases of the network using an optimization algorithm such as stochastic gradient descent.

### 10.3 Examples from our code

... Dokoncz Kacper ...

In summary, the chain rule is a powerful tool that enables us to efficiently compute gradients in neural networks, facilitating the training process and allowing networks to learn complex relationships in data.

## 11 Testing the network

napisz kilka slow o GUI, screeny wrzuc z tego ...



## **12 Conclusion**

This is the conclusion of the document.

## References

- [1] Mbali Kalirane. “Gradient Descent vs. Backpropagation: What’s the Difference?” In: *Data Science Blogathon* (Apr. 2023). Published as a part of the Data Science Blogathon. URL: <https://datahack.analyticsvidhya.com/blogathon/>.
- [2] Wikipedia contributors. “Feedforward Neural Network”. In: (). Accessed on February 6, 2024.



```

1  learning_rate = 0.8
2  batch_size = 100
3  numberOfSteps = 1000
4
5  training_data = []
6  for j in range(1000):
7      flattened_image = [pixel for sublist in images[j] for pixel in sublist]
8      training_data.append(DataPoint(flattened_image, labels[j], 10))
9  batches = []
10 for j in range(0, len(training_data), batch_size):
11     batches.append(training_data[j:j + batch_size])
12
13 print("Data loaded. Starting training...")
14
15 for i in range(numberOfSteps):
16     for j in range(len(batches)):
17         nn.learn(batches[j], learning_rate)
18         print("Step: ", i, "Batch: ", j, " Cost: ", nn.total_cost(batches[j]))
19         plt.pause(0.01)
20     nn.save_model('model.pkl')
21     print("Step: ", i, " Cost: ", nn.total_cost(training_data))
22     learning_rate *= 0.95
23 plt.show()

```

Figure 9: Training a neural network using a simple form of gradient descent on the MNIST dataset.

**Learning rate** - determines the step size at each iteration of gradient descent and the speed at which we move down the slope. Learning rate plays important role in between optimization time and accuracy. Step size is measured by a parameter  $\alpha$ . Small  $\alpha$  means small step size, large  $\alpha$  means large step size. If its too large then it can jump through minimum of the function. This parameter needs to be optimized. High learning rate results in a higher step value and opposite.

**Batch size** refers to the number of training examples utilized in one iteration.

**The number of steps**, represents the total number of times the entire training dataset is passed forward and backward through the neural network. Too few steps may lead to underfitting, while too many steps may result in overfitting on the training data. The optimal number of steps depends on the complexity of the task and the dataset.

```

1 def backpropagate(self, data_point):
2     outputs = self.calculate_outputs(data_point.inputs)
3     deltas = [output - expected_output for output, expected_output in zip(outputs, data_point.expected_outputs)]
4
5     for i in reversed(range(len(self.layers))):
6         layer = self.layers[i]
7         inputs = data_point.inputs if i == 0 else self.layers[i - 1].calculate_outputs(data_point.inputs)
8         layer.calculate_gradients(inputs, deltas)
9         if i > 0 and i < len(self.layers) - 1:
10             deltas = np.dot(layer.weights, deltas) * af.sigmoid_derivative(layer.calculate_outputs(data_point.inputs))
11

```

Figure 10: The `backpropagate` function is part of the neural network training process.

```

1 def calculate_gradients(self, inputs, deltas):
2     self.cost_gradient_w = np.outer(inputs, deltas)
3     self.cost_gradient_b = np.array(deltas)

```

Figure 11: The `calculate_gradients` method computes the gradients needed for the weight and bias adjustments.

It represents how much the weights and biases should be changed to minimize the difference between the predicted and expected outputs. It's like figuring out the direction and magnitude of the correction needed to improve the network's performance.