Introduction to Machine Learning

HANDS-ON

CODING A FULL NEURAL NETWORK FROM SCRATCH USING PYTHON

DEPARTAMENTO DE INGENIERÍA TELEMÁTICA Y ELECTRÓNICA

UNIVERSIDAD POLITÉCNICA DE MADRID

EDUARDO JUÁREZ
ALEJANDRO MARTÍNEZ DE TERNERO

Contents

1. Introduction

Welcome to this hands-on where you will dive deep into the fascinating world of neural networks. Through this document, you will learn the fundamental concepts of neural networks by implementing one from scratch using Python. This practice is designed to enhance your understanding of how neural networks work and to give you practical experience in coding them.

By the end of this exercise, you will have built a fully functional neural network that can learn from data using backpropagation algorithm and the gradient descent optimization technique and make predictions on new data.

This exercise is divided into several sections, each covering a different aspect of neural networks. You will start by building a single neuron (perceptron) and then extend it to a multi-layer neural network. You will learn how to initialize the weights and biases, implement forward and backward propagation, and train the network using gradient descent.

By the end of this exercise, you will also be able to apply this neural network to a variety of areas like regression, classification or to approximate complex functions.

Let's get started!

2. Previous considerations

The exercises proposed in this document consist of a series of small programs written in Python, intended to get familiar with the programming language while building a fully-functional neural network from scratch. To implement the code of this document, the student can use:

1) Visual Studio IDE, installed in the virtual machine available in the Moodle's course site. The installed package already has Python installed and it can be straightforwardly used,

and

2) Google Colab. Although the use of the previous option is recommended, in case of difficulties with the virtual machine or if you want to make specific tests out of the normal working environment, you can use the Google's online coding tool, Colab [1].

Please note that the Virtual Machine password is: ***imlstudent2022***

3. Hands-on: Coding a neural network from scratch using Python

As the title of this document implies, we are going to develop our own neural network from scratch. The only dependencies we need are:

- **Numpy**: which is a scientific computing kit with a handful of operations on arrays [2].
- **Matplotlib:** which we will only use in order to represent some results [3].

- **Scikit-learn:** Is a handy toolking which we will use to get datasets [4].

In case you don't have these packages installed, you can do so with the following code:

```
pip3 install numpy matplotlib scikit-learn
```

a) Simple start: a single artificial neuron

Remember the perceptron? It's the simplest unit of computation inside a neural network. It takes a set of inputs, multiplies them by a set of weights, sums them up, and passes the sum through an activation function. The output is the result of this function.

Starting with just the neuron, without the activation function, we can write its inner working as:

$$y = w_1 * x_1 + w_2 * x_2 + \cdots + w_n * x_n + b$$

Where y is the output, w_i are the weights, b is the bias term, and x_i are the inputs.
Of course, we can reduce it into a formal mathematical formulation:

$$y = \sum_{i}^{n} w_i \cdot x_i + b$$

And when we apply the activation function, we get the following:

$$y = f(\sum_{i}^{n} \omega_i \cdot x_i + b)$$

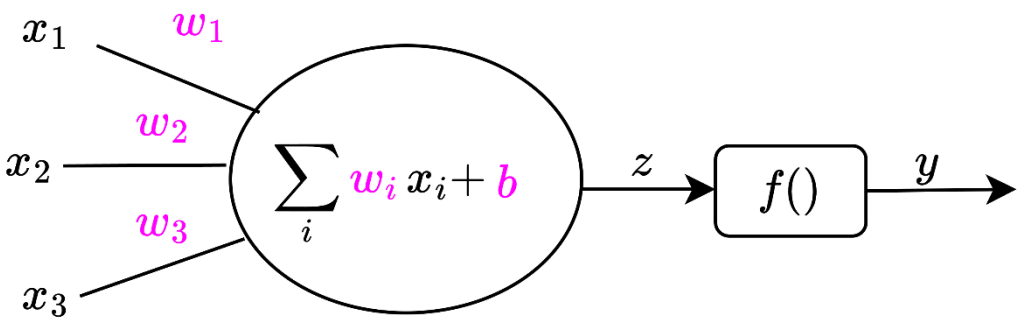


Figure 1. Artificial neuron. In Pink the parameters that are learned.

We can code this into a simple Python class, in which we expose a method (*forward*) that will take the inputs x_i and the activation function $f(.)$ and produce the output y.

Remember that the weights and bias are the **learnable parameters** of this neuron. This means that they will be updated in order to represent the given dataset.

```
 1. import numpy as np  # NumPy is the fundamental package for scientific computing with
Python
 2. import matplotlib.pyplot as plt  # Matplotlib is a Python plotting library which produces
publication quality figures.
 3.
 4. class Neuron:
 5.     def __init__(self, n_inputs, activation_function):
 6.         """
 7.         Initializes the neuron with random weights and bias, and sets
```

```
 8.         the activation function.
 9.         Parameters:
10.             n_inputs (int): The number of input connections to the neuron.
11.             activation_function (callable): The activation function to be
12.                 used by the neuron.
13.         """
14.         self.weights = np.random.uniform(-1, 1, n_inputs)
15.         self.bias = np.random.uniform(-1, 1)
16.         self.activation_function = activation_function
17.
18.     def forward(self, inputs):
19.         """
20.         Perform the forward pass of the neural network layer.
21.         Parameters:
22.             inputs (numpy.ndarray): Input data to the layer.
23.         Returns:
24.             numpy.ndarray: Output of the layer after applying weights, bias,
25.                 and activation function.
26.         """
27.
28.         self.inputs = inputs
29.         z = np.dot(self.weights, inputs) + self.bias
30.         self.output = self.activation_function(z)
31.         return self.output
```

In lines 14 and 15, we can see that the weights are initialized with a random *uniform* distribution in the range (-1, 1), as well as the bias. As you can imagine, for the weights we need one per each input of the neuron, while the bias is a single value.

You may be wondering… where is the *sum* operation of the neuron? Well, if we take a look at the documentation on *np.dot* [5] we can see that it is implicit in this operation.

Exercise 1: Knowing the operations that are needed in the neuron, substitute the code in line 29 with the code without using NumPy operations.

```

```

Defining an activation function

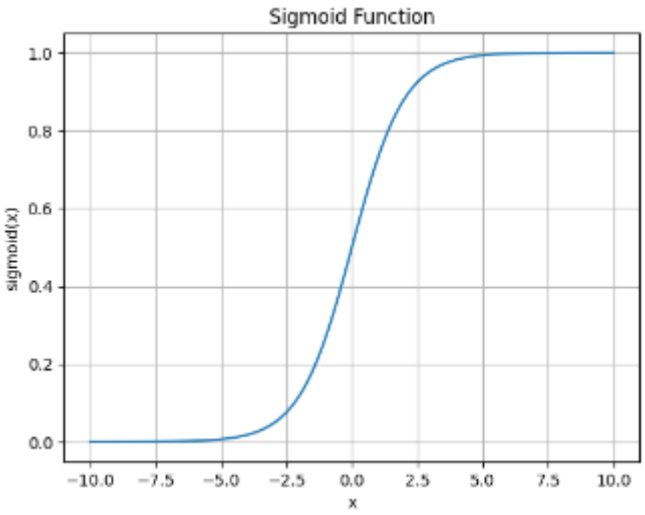
There is one last thing that we need to define. In the constructor for the Neuron class, we can see that we need to use an **activation function**. Remember that this activation function (also called squashing function) is responsible for scaling the output values between a given range. As an example, we are going to build the sigmoid activation function.

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

```
1. def sigmoid(x):
2.     return 1 / (1 + np.exp(-x))
```

Let's plot this function, so we can visualize what is really doing with the data.

```
1. x = np.linspace(-10, 10, 100)  # Create 100 points between -10 and 10
2. y = sigmoid(x)  # Apply the sigmoid function to each point
3. plt.plot(x, y)  # Plot x and y
4. plt.xlabel("x")
5. plt.ylabel("sigmoid(x)")
6. plt.title("Sigmoid Function")
7. plt.grid()
8. plt.show()
```



So, the sigmoid function will squash the output between 0 and 1. There is a clear *"linear"* part (in the figure from -2 to 2 aprox) and we also find two saturation areas at the borders, where the values are squashed close to 0 or 1.

Exercise 2: Search for 3 other activation functions online, implement them and plot them against the sigmoid function. Can you identify when one function is best than another.

Nice, so we have a working neuron! Let's try to play with it.

We are going to define a neuron with two inputs and a sigmoid activation function, then we would like to test the forward method.

```
neuron = Neuron(n_inputs=2, activation_function=sigmoid)

# Let's test the neuron with some inputs
input = [1, 2]
output = neuron.forward(input)
print(f"Input is {input}, output is {output}")

# Let's test the neuron with some other inputs
input = [1, -2]
output = neuron.forward(input)
print(f"Input is {input}, output is {output}")
```

```
Input is [1, 2], output is 0.6346325032155706
Input is [1, -2], output is 0.31907088859242594
```

Exercise 3: Run the above code several times. Why the output is different for the same output? Without modifying the Neuron class, can you

Well, the weights and bias are initialized randomly, so each time a new Neuron object is created, new weights and bias are generated. Thus, the result depends on this random initialization.

b) Create a neural network

Remember, the structure of a neural network is composed of layers of neurons where the output of one layer is the input of the next. So, we can define a neural network as a list of layers, where each layer is a list of neurons!

Neural network structure

- **Input layer:** This layer consists of the input data itself. If we have (n) features, the input layer will have (n) receptors (inputs).
- **Hidden layers**: These layers perform most of the computations required by the network. Each neuron in a hidden layer receives input from all neurons in the previous layer and sends its output to all neurons in the next layer.
- **Output layer:** This layer produces the final output of the network. The number of neurons in this layer depends on the type of task (e.g., for binary classification, we typically have one neuron with a sigmoid activation function).

Example network configuration and visualization

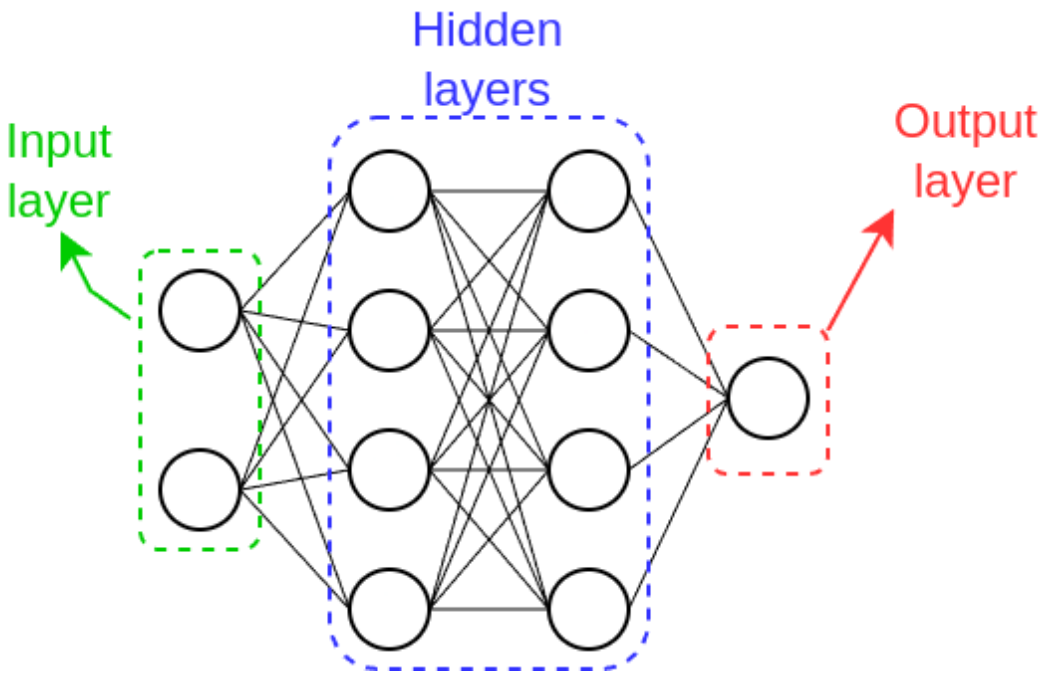


Figure 2. Example neural network

- **Input Layer:** 2 inputs (features).
- **First Hidden Layer:** 4 neurons, each with 2 inputs.
- **Second Hidden Layer:** 4 neurons, each with 4 inputs (outputs from the first hidden layer)
- **Output Layer:** 1 neuron, with 4 inputs (outputs from the second hidden layer)

Let's define each layer as a list of neurons.

```
1. n_inputs = 2
2.
3. hidden_layer_1 = [Neuron(n_inputs=n_inputs, activation_function=sigmoid) for _ in
range(4)]
4. hidden_layer_2 = [Neuron(n_inputs=4, activation_function=sigmoid) for _ in range(4)]
5. hidden_layers = [hidden_layer_1, hidden_layer_2]
6.
7. output_layer = [Neuron(n_inputs=4, activation_function=sigmoid)]
```

How to code the forward pass

We already have a list of hidden layers, composed each one with neurons. We can do a loop simulating the *forward* step of a neural network. For this, input data must flow from the first

hidden layer to the output layer. Use the following code snippet, and make sure you understand the flow of information between the neurons.

```
 1.  input_data = np.array([1, 2])
 2.
 3. for i, layer in enumerate(hidden_layers):
 4.     print(f"-> Input data for the layer {i} is: {input_data}")
 5.     input_data = np.array([neuron.forward(input_data) for neuron in layer])
 6.     print(f"<- Output data for the layer {i} is: {input_data}")
 7.
 8. print(f"-> Input data for the output layer is: {input_data}")
 9. output_data = np.array([neuron.forward(input_data) for neuron in output_layer])
10. print(f"<- Output data for the output layer is: {output_data}")
11.
```

```
-> Input data for the layer 0 is: [1 2]
<- Output data for the layer 0 is: [0.09022574 0.6179164  0.13711332 0.68847486]
-> Input data for the layer 1 is: [0.09022574 0.6179164  0.13711332 0.68847486]
<- Output data for the layer 1 is: [0.42163333 0.44909666 0.51927815 0.4261581 ]
-> Input data for the output layer is: [0.42163333 0.44909666 0.51927815 0.4261581 ]
<- Output data for the output layer is: [0.55422363]
```

Does this make sense? If we take a look at the network diagram above, we are inputting two values, which will be 4 output values from the first hidden layer (one per each neuron), 4 values for each neuron in the second hidden layer and the output layer just takes the 4 inputs from the last neurons and produces a single output.

So, knowing that our neural network will have a series of hidden layers, as well as an output layer and that we will need to forward the input data layer by layer, we can code this into a class that holds all the layers and a forward method to do the forward pass.

In order to organize the code, we will implement this as:

```
 1. class NeuralNetwork:
 2.     def __init__(self, hidden_layers, output_layer):
 3.         """
 4.         Initializes a neural network with the given architecture.
 5.         Args:
 6.             hidden_layers (list[list[Neuron]]): A list of Neuron layers.
 7.             output_layer (list[Neuron]): The output layer of the neural network.
 8.         """
 9.         self.hidden_layers = hidden_layers
10.         self.output_layer = output_layer
11.
12.     def forward(self, inputs):
13.         """
14.         Perform the forward pass of the neural network.
15.         Args:
16.             inputs (numpy.ndarray): Input data for the forward pass.
17.         Returns:
18.             numpy.ndarray: The output of the neural network.
19.         """
20.         for layer in self.hidden_layers:
21.             inputs = np.array([neuron.forward(inputs) for neuron in layer])
22.         final_outputs = np.array([neuron.forward(inputs) for neuron in self.out-
put_layer])
23.         return final_outputs
24.
```

We are using a reduction of the loop in lines 21 and 22. This is a Python feature called *list comprehension* which allow us to reduce the code size, while improving performance. You can find more information online [6]. It is equivalent of using the normal for loop.

We can see that the class that we have created is equivalent to the implementation using for loops with the following code:

```
num_inputs = 2
hidden_layers = [[Neuron(2, sigmoid), Neuron(2, sigmoid),Neuron(2, sigmoid), Neuron(2, sigmoid)]]
output_layer = [Neuron(4, sigmoid)]

# Code using for-loop
inputs = np.array([1, 2])
for i, layer in enumerate(hidden_layers):
    inputs = np.array([neuron.forward(inputs) for neuron in layer])

output_data = np.array([neuron.forward(inputs) for neuron in output_layer])
print(f"For-loop approach output: {output_data}")

# Code using NeuralNetwork class
inputs = np.array([1, 2])
neural_net = NeuralNetwork(hidden_layers, output_layer)
output = neural_net.forward(inputs)
print(f"Neural network output: {output}")
```

```
For-loop approach output: [0.27758912]
Neural network output: [0.27758912]
```

c) Backpropagation on a neuron

In the forward pass, we passed the input data through the network, layer by layer, and obtained the output. Now, we need to do the opposite: we need to pass the error back through the network, layer by layer, and update the weights and biases.

With the network that we have defined (with 2 hidden layers and 1 output layer), we need to calculate the error of the output layer and propagate it back to the second hidden layer, then to the first hidden layer, and finally to the input layer.

The error propagated back (backpropagated) is calculated as the derivative of the loss function with respect to the weights and biases of the network, and it is used to update them.

Steps for backpropagation

1. **Calculate the Error at the Output Layer**: Compute the difference between the predicted output and the actual target. This error is then used to calculate the gradient of the loss function with respect to the output.
2. **Propagate the error backwards**: For each layer, starting from the output layer and moving backwards to the input layer. Calculate the gradient of the loss function with respect to the weights and biases.
3. **Update the weights and biases**: Adjust the weights and biases in the direction that reduces the error.

How are weights and bias of a neuron updated?

Each neuron will produce an output, and that output will have an associated error (the higher the error, the worse the output represents the problem at hand). We would like to find **how much** each weight and bias contributed to the error, so we can update them accordingly.

And... how can we mathematically determine the contribution of each weight and bias to the error? We can use the chain rule!

The chain rule states that the derivative of a composition of functions is the **product of the derivatives of each function**. This is represented in the following figure:

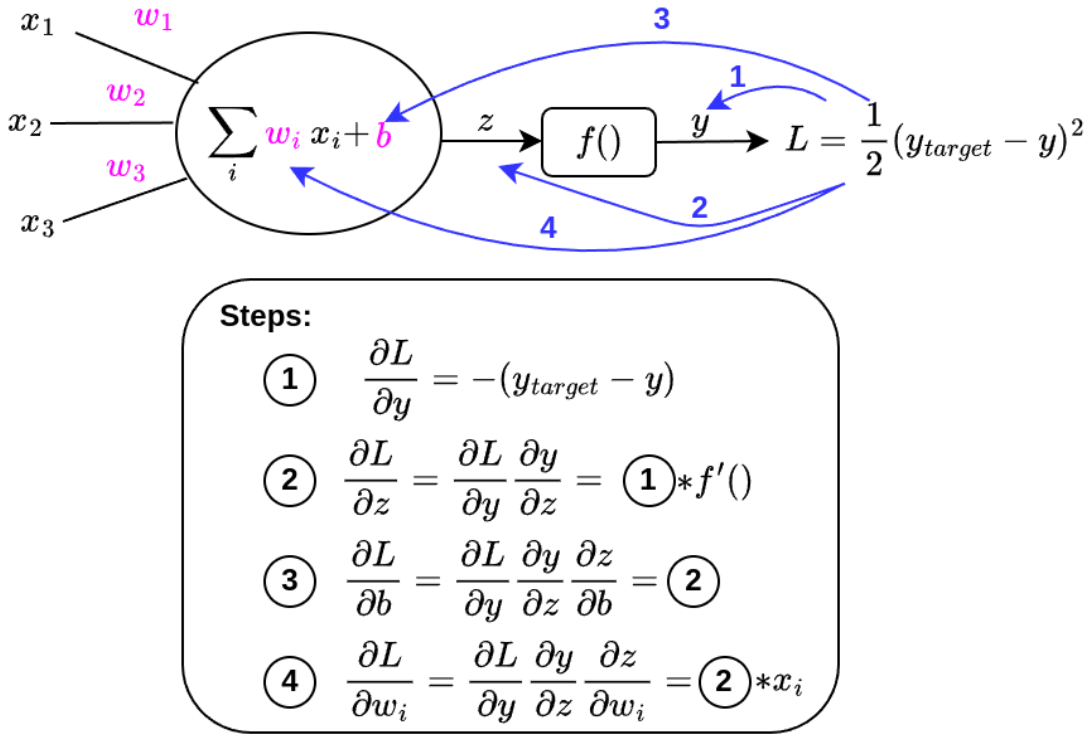


Figure 3. Backward pass on a single neuron.

Once we have determined how to update the weights, we see that we need the derivative of the activation function $f'(x)$. Computing the derivative of the sigmoid function we end up with the following expression:

$$\frac{df(x)}{dx} = \frac{e^{-x}}{(1+e^{-x})^2} = f(x)(1-f(x))$$

However, looking at the figure, we can see that the input of the derivative *y* is the output of the activation function. We can then write a function that takes the output of a sigmoid and returns the derivative as:

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

Remember that *x* is, in this case, the output of a sigmoid function!

Exercise 4: Recover the other activation functions that you have coded in the exercise 2 and code their respective derivatives in the same manner. (The input is the output of the same function)

With this, we can improve our Neuron class and add the backpropagation part in which we compute the derivatives:

```
1. class Neuron:
2.     def __init__(self, num_inputs, activation_function, activation_derivative):
3.         self.weights = np.random.uniform(-1, 1, num_inputs)
4.         self.bias = np.random.uniform(-1, 1)
5.         self.output = 0
6.         self.inputs = 0
7.         self.activation_function = activation_function
8.         self.activation_derivative = activation_derivative
```

```
 9.
10.     def forward(self, inputs):
11.         self.inputs = inputs
12.         z = np.dot(inputs, self.weights) + self.bias
13.         self.output = self.activation_function(z)
14.         return self.output
15.
16.     def backpropagate(self, d_error, learning_rate):
17.         # d_error is the gradient of the error w.r.t the output of the neuron, d_1 in
the previous picture
18.         d_2 = d_error * self.activation_derivative(self.output)
19.         d_3 = d_2
20.         d_4 =  d_2 * self.inputs
21.
```

Ok but… how are the weights and bias updated?

For that, we can use the **gradient descent** algorithm. It's a simple algorithm that updates the weights and biases in the opposite direction of the gradient of the error with respect to the weights and biases. Meaning that if the error is increasing with respect to a weight, we should decrease that weight, and if the error is decreasing with respect to a weight, we should increase that weight.

The update rule for the weights and biases is:

$$w_i = w_i - \alpha \frac{\partial \mathrm{L}}{\partial \mathrm{w_i}}$$

$$b = b - \alpha \frac{\partial L}{\partial b}$$

Where:

- w_i are the weights.
- b is the bias
- α is the learning rate.
- $\frac{\partial L}{\partial w_i}, \frac{\partial L}{\partial b}$ are the partial derivatives of the error (loss) with respect to the weights and bias, respectively.

We can also include this piece in the code for the backpropagation of our Neuron, which will now compute the derivatives and update its own weights and the bias:

```
 1. class Neuron:
 2.     def __init__(self, num_inputs, activation_function, activation_derivative):
 3.         self.weights = np.random.uniform(-1, 1, num_inputs)
 4.         self.bias = np.random.uniform(-1, 1)
 5.         self.output = 0
 6.         self.inputs = 0
 7.         self.activation_function = activation_function
 8.         self.activation_derivative = activation_derivative
 9.
10.     def forward(self, inputs):
11.         self.inputs = inputs
12.         total = np.dot(inputs, self.weights) + self.bias
13.         self.output = self.activation_function(total)
14.         return self.output
15.
16.     def backpropagate(self, d_error, learning_rate):
17.         # d_error is the gradient of the error w.r.t the output of the neuron
18.         d_2 = d_error * self.activation_derivative(self.output)
19.         d_3 = d_2
20.         d_4 =  d_2 * self.inputs
21.
22.         # Update the weights and bias
```

```
23.         self.weights -= learning_rate * d_4
24.         self.bias -= learning_rate * d_3
```

But wait, what is the learning rate and why do the weights and bias update with a *minus* sign?

The learning rate is a **hyperparameter** that controls how much we update the weights and biases. It's a small value that prevents the weights and biases from being updated too much in one iteration. If the learning rate is too high, the weights and biases can oscillate around the minimum of the error function (third figure), and if it's too low (first figure), the weights and biases can take too long to converge to the minimum. We can see the intuition behind how this works with the following figure.

Figure 4. Learning rate parameter. Source [7]

The meaning of the word **hyperparameter** is that it's a parameter that is not learned by the model, but set by the user. It's a parameter that controls the learning process, and multiple values must be tested to find the best one that fits the problem at hand.

On the other hand, the negative sign in the update rule is because we want to move in the opposite direction of the gradient of the error. If the error is increasing with respect to a weight, we should decrease that weight, and if the error is decreasing with respect to a weight, we should increase that weight.

With the image above, we can see that we want to move in the opposite direction of the gradient of the error, we want to move towards the minimum of the error function!

How could the neuron propagate the error to the others?

When a neuron is connected to multiple neurons in the next layer, we need to **sum the errors of all the neurons** in the next layer to get the error of the current neuron. This is because the error of the current neuron is the sum of the errors of the neurons in the next layer, weighted by the weights connecting them.

This is just **propagation** of error, and for that, we need one more derivative: the derivative of the output of the neuron with respect to the input of the neuron. This is the value that we will propagate back to the previous layer.

Mathematically, this is:

$$\frac{\partial L}{\partial x_i} = \frac{\partial L}{\partial y}\frac{\partial y}{\partial x_i}$$

And it is the yellow step (number 5) that we see in the following figure:

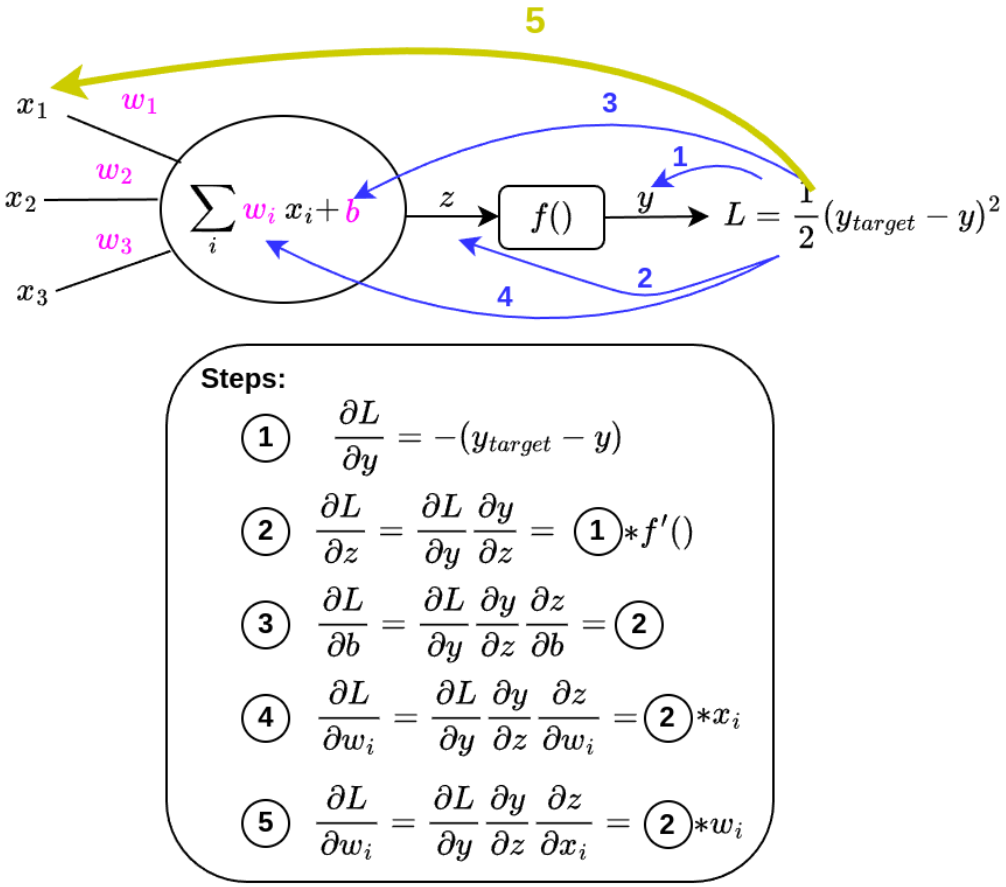


Figure 5. Full backpropagation pass on an artificial neuron.

Implementation is easy because we already have all the derivatives that we will need, we can implement this as follows:

```
class Neuron:
    def __init__(self, num_inputs, activation_function, activation_derivative):
        self.weights = np.random.uniform(-1, 1, num_inputs)
        self.bias = np.random.uniform(-1, 1)
        self.output = 0
        self.inputs = 0
        self.activation_function = activation_function
        self.activation_derivative = activation_derivative

    def forward(self, inputs):
        self.inputs = inputs
        total = np.dot(inputs, self.weights) + self.bias
        self.output = self.activation_function(total)
        return self.output

    def backpropagate(self, d_error, learning_rate):
        # d_error is the gradient of the error w.r.t the output of the neuron
        d_2 = d_error * self.activation_derivative(self.output)
        d_3 = d_2
        d_4 =  d_2 * self.inputs

        # Update the weights and bias
        self.weights -= learning_rate * d_4
        self.bias -= learning_rate * d_3

        return d_2 * self.weights
```

d) Backpropagation on the full network

For the network, remember that in the forward pass we will pass the input data through the network, layer by layer, and get the output.

The backpropagation is the same, but we will do it in **reverse** and propagating the error instead of the input value. We will calculate the error of the output layer and propagate it back to the second hidden layer, then to the first hidden layer, and finally to the input layer.

We can represent a simple example first:

Let's consider a simple neural network with one hidden layer and one output layer, each with 1 neuron.

Forward Pass:

- Input: *x*
- Hidden Layer: $h = \mathrm{f}(w_1 \cdot x + b_1)$
- Output Layer: $y = \mathrm{f}(w_2 \cdot h + b_2)$

Backpropagation:

- Compute the error: $\mathrm{L} = \mathrm{loss}(\mathrm{y},\mathrm{target})$
- Calculate gradients: ($\frac{\partial E}{\partial W_2}$), ($\frac{\partial E}{\partial b_2}$), ($\frac{\partial E}{\partial W_1}$),($\frac{\partial E}{\partial b_1}$)
- Update weights and biases:($W_2 = W_2 - \eta \frac{\partial L}{\partial W_2}$), ($b_2 = b_2 - \alpha \frac{\partial L}{\partial b_2}$), ($W_1 = W_1 - \alpha \frac{\partial L}{\partial W_1}$), ($b_1 = b_1 - \alpha \frac{\partial L}{\partial b_1}$).

With a more complex neural network, we will have something like the following figure:

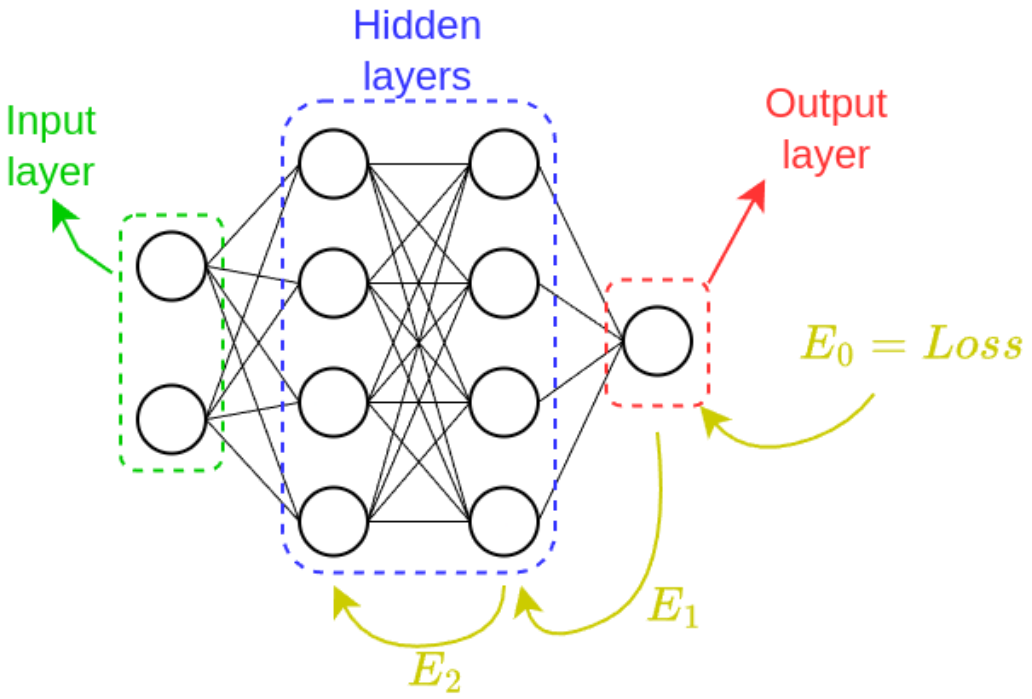


Figure 6. Error backpropagation in the neural network.

The code for the backpropagation of the whole neural network is somewhat similar to the one in the Neuron class:

```
 1. class NeuralNetwork:
 2.     def __init__(self, hidden_layers, output_layer):
 3.         self.hidden_layers = hidden_layers
 4.         self.output_layer = output_layer
 5.
 6.     def forward(self, inputs):
 7.         for layer in self.hidden_layers:
 8.             inputs = np.array([neuron.forward(inputs) for neuron in layer])
 9.         final_outputs = np.array([neuron.forward(inputs) for neuron in self.out-
put_layer])
10.         return final_outputs
11.
12.     def backpropagate(self, inputs, targets, learning_rate):
13.         # Compute the output of the neural network
14.         final_outputs = self.forward(inputs)
15.         # Calculate the error as the gradient (target - output)
16.         d_error_d_output = -(targets - final_outputs)
17.         # Backpropagate through the output layer
18.         hidden_errors = [
19.             neuron.backpropagate(d_error_d_output[i], learning_rate) for i, neuron in
enumerate(self.output_layer)
20.         ]
21.
22.         # Add the errors of the output layer
23.         hidden_errors = np.sum(hidden_errors, axis=0)
24.
25.         # Backpropagate through the hidden layer
26.         for layer in reversed(self.hidden_layers):
27.             hidden_errors = np.sum(
28.                 [neuron.backpropagate(hidden_errors[i], learning_rate) for i, neuron
in enumerate(layer)], axis=0
29.             )
```

Essentially, line 14 computes the forward pass, line 16 the error with respect to the true value. This error is propagated neuron by neuron and layer by layer, which will update their weights.

Exercise 5: With the current NeuralNetwork and Neuron classes, create a reproducible example of a neural network with 1 input, 2 neurons in 1 hidden layer, and 1 neuron as output. The weights and bias of each neuron must be fixed. Make one backward pass and see what happens with the initial weights and biases. For the weights, they must be 0.5 and all bias to 0. Input is [1], and learning rate is 0.8.

e) Last step, training loop

The training of the network is the process of updating the weights and biases of the network using the backpropagation algorithm. We will do this by iterating over the training data multiple times (epochs) and updating the weights and biases of the network using the gradients of the error with respect to the weights and biases.

The following code will train the network and print the loss every *`print_every`* epochs. (lines 17 to 20 are the ones doing all the work, the other part serves as a nice visualization of the decreasing error as the training progresses).

```
 1. # This will make our experiments reproducible (not random)
 2. np.random.seed(42)
 3.
 4. def train(network, inputs, targets, epochs=10000, learning_rate=0.1,
print_every=1000):
 5.     """
 6.     Trains a neural network using backpropagation.
 7.     Args:
 8.         network (object): The neural network to be trained. Must have `backpropagate`
and `forward` methods.
 9.         inputs (list or np.ndarray): The input data for training.
10.         targets (list or np.ndarray): The target outputs corresponding to the input
data.
11.         epochs (int, optional): The number of training iterations. Default is 10000.
12.         learning_rate (float, optional): The learning rate for the backpropagation al-
gorithm. Default is 0.1.
13.         print_every (int, optional): The interval (in epochs) at which to print the
total error. Default is 1000.
14.     Prints:
15.         The total error at specified intervals during training.
16.     """
17.     for epoch in range(epochs):
18.         # Train on each input-output pair
19.         for input_data, target in zip(inputs, targets):
20.             network.backpropagate(input_data, target, learning_rate)
21.
22.         # Every once in a while, print the total error to see progress (we shoud see
it decrease)
23.         if epoch % print_every == 0:
24.             total_error = 0
25.             for input_data, target in zip(inputs, targets):
26.                 predicted_output = network.forward(input_data)
27.                 total_error += np.sum((target - predicted_output) ** 2)
28.             print(f"Epoch {epoch} - Total Error: {total_error:.4f}")
```

4. Evaluating the neural network

a) Neural networks as universal function approximators

Neural networks are known to be **universal function approximators**. This means that they can approximate **any** function, no matter how complex it is, given enough neurons and layers.

We can play with this concept by recreating some functions using our network!

XOR function

The XOR (exclusive OR) function is a binary operation that takes two binary inputs and produces a single binary output. It is defined by the following conditions:

- The output is **true (1)** if the two inputs are **different**.

- The output is **false (0)** if the two inputs are the **same**.

The truth table for the XOR function is as follows:

Input 1	Input 2	Output
0	0	0
0	1	1
1	0	1
1	1	0

Let's create some inputs and targets for this function (we essentially need to code the truth table).

```
# Define the data of the function that we want to learn
inputs = np.array([[0, 0], [0, 1], [1, 0], [1, 1]])
targets = np.array([[0], [1], [1], [0]])
```

We can define a simple network for this task, for example, 2 hidden layers with 2 neurons each, and an output layer with 1 neuron that will give us the result.

```
# Example network architecture: 2 inputs, 2 hidden layers with 2 neurons each, 1 output neuron
num_inputs = 2
hidden_layers = [
    [
        Neuron(2, sigmoid, sigmoid_derivative),
        Neuron(2, sigmoid, sigmoid_derivative),
    ]
]
output_layer = [Neuron(2, sigmoid, sigmoid_derivative)]

network = NeuralNetwork(hidden_layers=hidden_layers, output_layer=output_layer)
```

And just call our *train* method and display some predictions of the trained network.

```
train(network, inputs, targets, epochs=10000, learning_rate=0.5)

print("\nTesting the neural network after training:")
for input_data, target in zip(inputs, targets):
    predicted_output = network.forward(input_data)
    print(f"  Input: {input_data} - Predicted Output: {predicted_output} - Real Output: {target}")
```

```
Epoch 0 - Total Error: 1.0047
Epoch 1000 - Total Error: 0.7280
Epoch 2000 - Total Error: 0.6700
Epoch 3000 - Total Error: 0.0292
Epoch 4000 - Total Error: 0.0122
Epoch 5000 - Total Error: 0.0075
```

```
Epoch 6000 - Total Error: 0.0054
Epoch 7000 - Total Error: 0.0042
Epoch 8000 - Total Error: 0.0035
Epoch 9000 - Total Error: 0.0029

Testing the neural network after training:
  Input: [0 0] - Predicted Output: [0.02384224] - Real Output: [0]
  Input: [0 1] - Predicted Output: [0.97261673] - Real Output: [1]
  Input: [1 0] - Predicted Output: [0.97321771] - Real Output: [1]
  Input: [1 1] - Predicted Output: [0.02207986] - Real Output: [0]
```

Lines 3 to 6 in the above code section just use the trained neural network to generate new predictions.

Exercise 6: Use the neural network to approximate the **AND** function now. Modify the code so that the output of the network is composed of 4 neurons, thus outputting an array with 4 positions, where the correct one has higher output than the others. This is called one-hot encoding [Link] and we will use this quite frequently with other examples.

Quadratic function

We can use a more complex function in a real scenario. For this, we will use the following:

$$y = x_1^2 + x_2^2$$

For this purpose, we are going to compute 35 random points of this function, and we will use those points to train the network.

```
# Define the quadratic function: y = x1^2 + x2^2
def quadratic_function(x):
    return np.sum(np.square(x), axis=1, keepdims=True)

number_of_training_points = 35
number_of_testing_points = 40

# Generate training data
inputs = np.random.uniform(-1, 1, (number_of_training_points, 2))  # 35 random points in 2D space
targets = quadratic_function(inputs)

# Example network architecture: 2 inputs, 1 hidden layer with 4 neurons, 1 output neuron
hidden_layers = [
    [Neuron(2, sigmoid, sigmoid_derivative) for _ in range(4)],
    [Neuron(4, sigmoid, sigmoid_derivative) for _ in range(4)],
]
output_layer = [Neuron(4, sigmoid, sigmoid_derivative)]

network = NeuralNetwork(hidden_layers=hidden_layers, output_layer=output_layer)
train(network, inputs, targets, epochs=4500, learning_rate=0.1, print_every=1000)
```

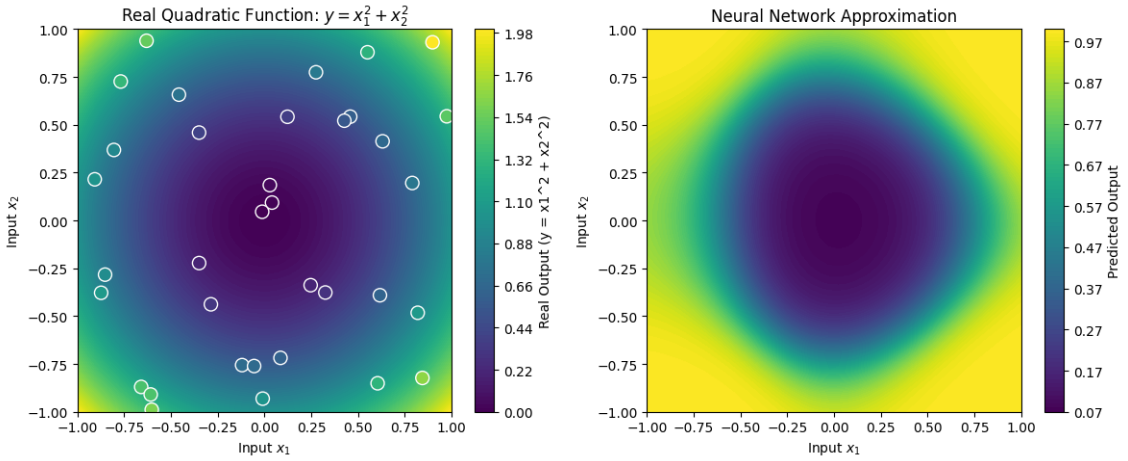
In order to test it, we are going to create a uniform grid of points (for example, 40 points), and then we will plot the real function and the approximated by the neural network one. The left plot will also show which points have been used in the training.

```
# Test the network on a grid of points and plot the results
x_values = np.linspace(-1, 1, number_of_testing_points)
y_values = np.linspace(-1, 1, number_of_testing_points)
xx, yy = np.meshgrid(x_values, y_values)
test_inputs = np.c_[xx.ravel(), yy.ravel()]

```

```
 7. # Get real function outputs
 8. real_outputs = quadratic_function(test_inputs)
 9.
10. # Get predicted outputs from the neural network
11. predicted = np.array([network.forward(input_data) for input_data in test_inputs])
12. predicted = predicted.reshape(xx.shape)
13.
14. plt.figure(figsize=(12, 6))
15.
16. # Plot real function
17. plt.subplot(1, 2, 1)
18. plt.contourf(xx, yy, real_outputs.reshape(xx.shape), levels=100, cmap="viridis")
19. plt.colorbar(label="Real Output (y = x1^2 + x2^2)")
20. plt.scatter(inputs[:, 0], inputs[:, 1], c=targets, cmap="viridis", edgecolors="w",
s=100)
21. plt.title("Real Quadratic Function: $y = x_1^2 + x_2^2$")
22. plt.xlabel("Input $x_1$")
23. plt.ylabel("Input $x_2$")
24.
25. # Plot approximated function
26. plt.subplot(1, 2, 2)
27. plt.contourf(xx, yy, predicted, levels=100, cmap="viridis")
28. plt.colorbar(label="Predicted Output")
29. plt.title("Neural Network Approximation")
30. plt.xlabel("Input $x_1$")
31. plt.ylabel("Input $x_2$")
32. plt.tight_layout()
33. plt.show()
34.
```

Lines 1 to 12 are the ones creating the test data and predicting with the trained network. The rest is used to visualize the solution.

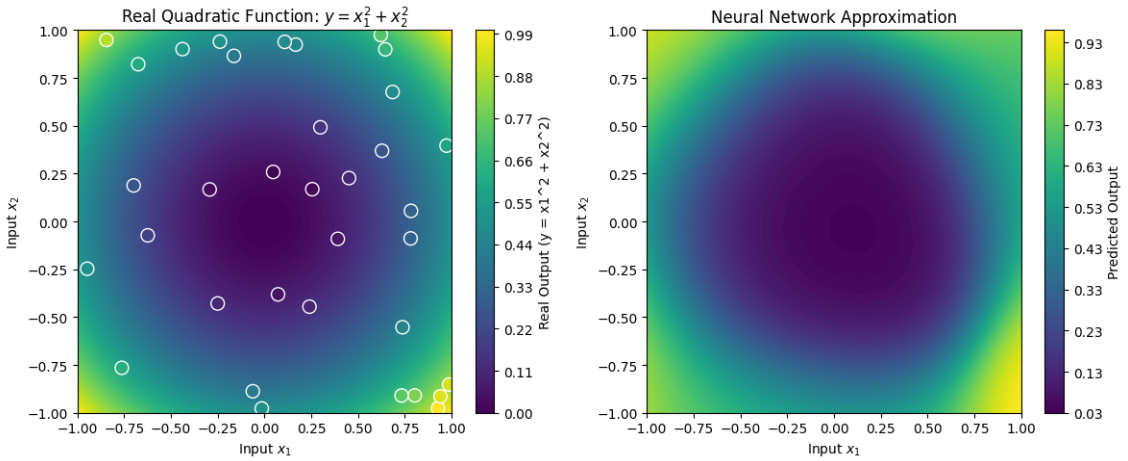


Why can't we reach a good accuracy? What is the problem with our network?

If we take a closer look, we can see that the output of the network is always between 0 and 1, which is not the case with the original function that we want to represent. This is because of the sigmoid activation function that we are using. The sigmoid function squashes the input into a range between 0 and 1.

Let's then do a normalization on the initial data, so the input data is between 0 and 1. This way, the output of the network will also be between 0 and 1. We will apply the following to the output of the quadratic function.

```
1. def normalize(x):
2.     return (x - np.min(x)) / (np.max(x) - np.min(x))
```



This is much better! Now, this is not a "complex" function, but only with 35 points we can have a simple NN that replicates a whole continuous function. Imagine that this function is highly complex, something that a computer needs some seconds or even minutes to process and that we train a NN with some points of this function. If we can achieve the network to learn the whole function, the time it takes to run a prediction with the network will be much less than the time taken to compute the function. This, as you can imagine, is of high interest as it can reduce complexity and time of computations in real-world problems.

b) Neural networks for regression tasks

Another nice task that can be solved using NNs is the regression. This implies learning from a set of data in order to predict the next value. This is used, for example, in time forecast, where a network learns from previous weather data in order to predict if tomorrow is going to be sunny or rainy.

We will now use our network to predict the house value in California districts [8], which can be used to predict the house price in California based on the house characteristics. It includes the following 8 features:

- **MedInc**: Median income in the district (scaled value).
- **HouseAge**: Median age of houses in the district.
- **AveRooms**: Average number of rooms per household.
- **AveBedrms**: Average number of bedrooms per household.
- **Population**: Total population in the district.
- **AveOccup**: Average number of occupants per household.
- **Latitude**: Latitude of the district.
- **Longitude**: Longitude of the district.

And the target variable is the median house value (in hundreds of thousands of dollars, eg: 4.5 means the house price is $450.000).

It is a huge dataset, so we are going to take a small set (lines 10-11 select 250 examples), normalize the information (here the StandardScaler from scikit-learn is used, which is a good resource that is often used in the Python machine learning community), and use a small network to predict the price of new houses. The network will be composed of a hidden layer with 8 neurons, and an output layer with 1 neuron returning the scaled median house value.

```
1. from sklearn.datasets import fetch_california_housing
2. from sklearn.model_selection import train_test_split
3. from sklearn.preprocessing import StandardScaler
4.
5. # California Housing Dataset
6. housing = fetch_california_housing()
```

```
data, targets = housing.data, housing.target

# Select a subset of the data
data = data[:250]
targets = targets[:250]

# Scale the data between 0 and 1
scaler = StandardScaler()
data = scaler.fit_transform(data)
targets = (targets - np.min(targets)) / (np.max(targets) - np.min(targets))

X_train, X_test, y_train, y_test = train_test_split(data, targets, test_size=0.2, random_state=42)

# Normalize features for better training
scaler = StandardScaler()
X_train = scaler.fit_transform(X_train)
X_test = scaler.transform(X_test)

hidden_layer = [Neuron(X_train.shape[1], sigmoid, sigmoid_derivative) for _ in range(8)]
output_layer = [Neuron(8, sigmoid, sigmoid_derivative)]

network = NeuralNetwork([hidden_layer], output_layer)
train(network, X_train, y_train, epochs=3000, learning_rate=0.03, print_every=500)

predicted_values = np.array([network.forward(x) for x in X_test])

plt.figure(figsize=(10, 5))
plt.scatter(y_test, predicted_values, color='blue', label='Predicted vs Actual')
plt.plot([y_test.min(), y_test.max()], [y_test.min(), y_test.max()], 'k--', linewidth=2, label='Perfect Prediction')
plt.xlabel('Actual Values')
plt.ylabel('Predicted Values')
plt.title('Neural Network Regression: California Housing')
plt.legend()
plt.show()
```

```
Epoch 0 - Total Error: 13.6394
Epoch 500 - Total Error: 1.6854
Epoch 1000 - Total Error: 1.2979
Epoch 1500 - Total Error: 1.1744
Epoch 2000 - Total Error: 1.1187
Epoch 2500 - Total Error: 1.0770
```

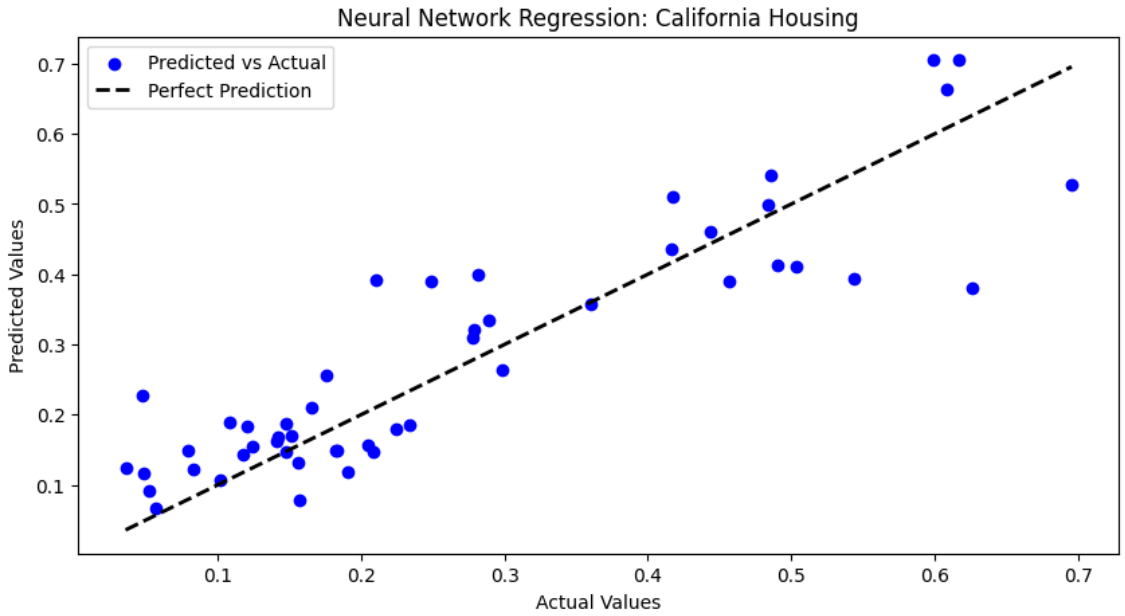


Figure 7. Example prediction on the California Housing dataset.

Exercise 7: Experiment with a more complex network. What are the results? Can you obtain a better regression? How does this affect performance?

c) Neural networks for classification tasks

Neural networks are also good for classification tasks. Remember that in these tasks we want to assign a predefined label to new data. In this case, we will use the Iris dataset [9]. This dataset has the objective of classifying between different species of flowers based on their morphological features, which are the sepal and petal length and width. The target are three species of the iris flower which are Setosa, Versicolor and Virginica.

We will use one-hot encoding to represent the data, so that the three classes will be:

- Setosa: [1,0,0]
- Versicolor: [0,1,0]
- Virginica: [0,0,1]

Again, we can get this dataset from scikit-learn:

```
1. from sklearn.datasets import load_iris
2. # Load the iris dataset
3. iris = load_iris()
4. X = iris.data
5. y = iris.target
```

Then, we can apply one-hot encoding, normalize and split between samples to train and to test the model:

```
 1. # Setosa: [1,0,0] , Versicolor: [0,1,0], Virginica: [0,0,1]
 2. y_multi = np.array([[1, 0, 0] if label == 0 else [0, 1, 0] if label == 1 else [0, 0,
1] for label in y])
 3.
 4. # Split dataset into training and testing sets
 5. X_train, X_test, y_train, y_test = train_test_split(X, y_multi, test_size=0.2, ran-
dom_state=42)
 6.
 7. # Standardize the features
 8. scaler = StandardScaler()
 9. X_train = scaler.fit_transform(X_train)
10. X_test = scaler.transform(X_test)
11.
```

And then, create the network, train and test:

```
 1. hidden_layers = [
 2.     [Neuron(4, sigmoid, sigmoid_derivative) for _ in range(4)],  # 4 neurons in the
hidden layer
 3. ]
 4. output_layer = [
 5.     Neuron(4, sigmoid, sigmoid_derivative),
 6.     Neuron(4, sigmoid, sigmoid_derivative),
 7.     Neuron(4, sigmoid, sigmoid_derivative),
 8. ]  # 3 output neurons
 9.
10. network = NeuralNetwork(hidden_layers=hidden_layers, output_layer=output_layer)
11. train(network, X_train, y_train, epochs=1000, learning_rate=0.1, print_every=100)
12.
13. predictions = []
14. for input_data, target in zip(X_test, y_test):
15.     predicted_output = network.forward(input_data)
16.     predictions.append(predicted_output)
17.     print(f"  Input: {input_data} - Predicted Output: {predicted_output} - Real Out-
put: {target}")
18.
19. def calculate_accuracy(predictions, targets):
20.     # Apply threshold to get binary predictions
```

```
    binary_predictions = (predictions > 0.5).astype(int)
    # Calculate accuracy
    accuracy = np.mean(np.all(binary_predictions == targets, axis=1))
    return accuracy

# Calculate accuracy
predictions = np.array(predictions)
accuracy = calculate_accuracy(predictions, y_test)
print(f"\nAccuracy of the neural network: {accuracy * 100:.2f}%")
```

5. Let's code improvements for our network

Although the network that we have built is fully functional, you may have noticed several things in which it can improve.

a) Weight initialization

The weights and biases of the network are initialized randomly, and this can lead to bad results. A bad initialization can lead to bad results, and a good initialization can lead to faster convergence and better results.

There are several ways to initialize the weights and biases of a neural network, some of the most common ones are:

- **Xavier initialization (also called Glorot initialization):** is best suitable for activation functions like sigmoid or tanh, where the variance of the weights needs to be kept in a specific range to maintain stability. Link to the paper [10]. The bias is usually initialized to zero.
- **He initialization:** is more appropriate for networks using ReLU or its variants because it takes into account the fact that ReLU neurons can "die" (output zero values when input is <0) Link to the paper [11]. (It contains also a new activation function called PReLU).
- **LeCun initialization:** is efective for the Leaky ReLU and similar activation functions. Link to the paper[12]. Can also work well with sigmoid and tanh activations.

```
def xavier_init(n_inputs):
    limit = np.sqrt(6 / n_inputs)
    return np.random.uniform(-limit, limit, n_inputs)

def he_init(n_inputs):
    stddev = np.sqrt(2 / n_inputs)
    return np.random.normal(0, stddev, n_inputs)

def lecun_init(n_inputs):
    stddev = np.sqrt(1 / n_inputs)
    return np.random.normal(0, stddev, n_inputs)
```

We can incorporate this into the Neuron class like:

```
class Neuron:
    def __init__(self, num_inputs, activation_function, activation_derivative,
init_method= "xavier"):
        if init_method == 'xavier':
            self.weights = xavier_init(num_inputs)
        elif init_method == 'he':
            self.weights = he_init(num_inputs)
        elif init_method == 'lecun':
            self.weights = lecun_init(num_inputs)
        else:
            # Uses the default initialization method (random uniform)
            self.weights = np.random.uniform(-1, 1, num_inputs)

        # Initialize bias randomly (could also use a small constant like 0.01)
        self.bias = np.random.uniform(0.001, 1)
```

```
15.         self.output = 0
16.         self.inputs = 0
17.         self.activation_function = activation_function
18.         self.activation_derivative = activation_derivative
19.
20.     def forward(self, inputs):
21.         self.inputs = inputs
22.         total = np.dot(self.inputs, self.weights) + self.bias
23.         self.output = self.activation_function(total)
24.         return self.output
25.
26.     def backpropagate(self, d_error, learning_rate):
27.         # d_error is the gradient of the error w.r.t the output of the neuron, d_1 in
the previous picture
28.         d_2 = d_error * self.activation_derivative(self.output)
29.         d_3 = d_2
30.         d_4 = self.inputs* d_2
31.
32.         # Update the weights and bias
33.         self.weights -= learning_rate * d_4
34.         self.bias -= learning_rate * d_3
35.
36.         return d_2 * self.weights
```

b) Improving stability and convergence

We can also tackle some of the things that we have not considered during training. Several of the things that are easily fixable are:

1. We are always inputting the data in the same order to the network. This can lead to a phenomenon called **overfitting**. In this case, the network will learn the data by heart, and will not be able to generalize to new data. We can shuffle the data before each epoch to avoid this. A situation like the one in the following image could be obtained, in which the NN is not really learning the intrinsic of data!
2. On the other hand, are using **gradient descent**, which is a simple algorithm that updates the weights and biases in the opposite direction of the gradient of the error with respect to the weights and biases. There are other algorithms that can improve the training process, like **Adam, RMSprop**, **Adagrad**, and **SGD with momentum**. These algorithms can improve the convergence of the network and make the training process faster. For our simple use case, we can improve gradient descent by implementing **momentum**. You have a nice explanation of how momentum helps in the following link [13]
3. Other thing to consider is the problem of **vanishing gradients**. This is a problem that occurs when the gradients of the error with respect to the weights and biases become very small, and the network stops learning. This can happen when the network is very deep, and the gradients are multiplied by the weights of the network, or when the activation function is not well chosen. An easy way to try to mend this is to use gradient clipping, which consists of clipping the gradients to a certain value if they become too large or too small. This can prevent the gradients from becoming too small and the network from stopping learning.
4. Usually, when training a neural network, we want to visualize the loss of the network over time. This can help us understand if the network is learning or not, and if it's converging to a good solution. We can plot the loss of the network over time using the `matplotlib` library. This is not only useful to know if the network is learning, but also **how** it is learning. A good visualization can help us understand if the network is converging to a good solution, if it's overfitting, or if it's underfitting. You have more information on how can we know these things in the following link [15].

To solve the problem 1 and 4, we can include this code in the training loop, so that it ends up like this:

```python
def train(network, inputs, targets, epochs=10000, learning_rate=0.1, print_every=1000, shuffle=True, plot=True):
    """
    Trains a neural network using backpropagation.
    Args:
        network (object): The neural network to be trained. Must have `backpropagate` and `forward` methods.
        inputs (list or np.ndarray): The input data for training.
        targets (list or np.ndarray): The target outputs corresponding to the input data.
        epochs (int, optional): The number of training iterations. Default is 10000.
        learning_rate (float, optional): The learning rate for the backpropagation algorithm. Default is 0.1.
        print_every (int, optional): The interval (in epochs) at which to print the total error. Default is 1000.
        shuffle (bool, optional): Whether to shuffle the data before each epoch. Default is True.
        plot (bool, optional): Whether to plot the results of the neural network after training. Default is True.
    Prints:
        The total error at specified intervals during training.
    """
    loss_history = []
    num_samples = len(inputs)
    total_error_last_epoch = 0
    for epoch in range(epochs):
        epoch_loss = 0
        # Shuffle data for each epoch
        if shuffle:
            indices = np.arange(num_samples)
            np.random.shuffle(indices)
            inputs, targets = inputs[indices], targets[indices]

        # Train on each input-output pair
        for input_data, target in zip(inputs, targets):
            predicted_output = network.forward(input_data)
            error = np.sum((target - predicted_output) ** 2)
            epoch_loss += error
            network.backpropagate(input_data, target, learning_rate)

        # Every once in a while, print the total error to see progress (we shoud see it decrease)
        if epoch % print_every == 0:
            total_error = 0
            for input_data, target in zip(inputs, targets):
                predicted_output = network.forward(input_data)
                total_error += np.sum((target - predicted_output) ** 2)
            if total_error == total_error_last_epoch:
                raise Exception(
                    f"Epoch {epoch} - Total Error: {total_error:.4f} - No improvement! Stopping training (maybe random error or bad initialization of the weights)"
                )
            print(f"Epoch {epoch} - Total Error: {total_error:.4f}")
            total_error_last_epoch = total_error
        loss_history.append(epoch_loss / num_samples)
    if plot:
        plt.plot(loss_history)
        plt.xlabel("Epoch")
        plt.ylabel("Loss")
        plt.title("Training Loss")
        plt.show()
```

Now, for solving problems 2 and 3, we could modify the Neuron class and incorporate the momentum and the clipping:

```python
class Neuron:
    def __init__(self, num_inputs, activation_function, activation_derivative, init_method= "xavier", momentum=0.5, clip_value=1.0):
        if init_method == 'xavier':
```

```
            self.weights = xavier_init(num_inputs)
        elif init_method == 'he':
            self.weights = he_init(num_inputs)
        elif init_method == 'lecun':
            self.weights = lecun_init(num_inputs)
        else:
            # Uses the default initialization method (random uniform)
            self.weights = np.random.uniform(-1, 1, num_inputs)

        # Initialize bias randomly (could also use a small constant like 0.01)
        self.bias = np.random.uniform(0.001, 1)
        self.output = 0
        self.inputs = 0
        self.activation_function = activation_function
        self.activation_derivative = activation_derivative
        # Momentum based variables
        self.momentum = momentum
        self.delta_weights = 0
        self.delta_bias = 0
        self.clip_value = clip_value

    def forward(self, inputs):
        self.inputs = inputs
        total = np.dot(inputs, self.weights) + self.bias
        self.output = self.activation_function(total)
        return self.output

    def backpropagate(self, d_error, learning_rate):
        # d_error is the gradient of the error w.r.t the output of the neuron, d_1 in
the previous picture
        d_2 = d_error * self.activation_derivative(self.output)
        d_3 = d_2
        d_4 =  d_2 * self.inputs

        d_4 = np.clip(d_4, -self.clip_value, self.clip_value)
        d_3 = np.clip(d_3, -self.clip_value, self.clip_value)

        # Update the weights and bias
        self.delta_weights = self.momentum * self.delta_weights - learning_rate * d_4
# It is also called the velocity
        self.weights += self.delta_weights
        self.delta_bias = self.momentum * self.delta_bias - learning_rate * d_3  # It
is also called the velocity
        self.bias += self.delta_bias

        return d_2 * self.weights
```

c) Performance optimizations

You may have noticed that out current implementation is pretty slow. This is because of two things:

- On the one hand, our network loops every neuron sequentially. Usually this is not the case for when we are concerned with performance. If, instead of defining individual neurons, we define layers, we could do all the forward and backwards operation of one layer as a matrix multiplication. This reduces a lot the computation cycles needed to run our network.
- On the other hand, we are looping through the dataset inputting one value at a time into the network. Again, this is usually not the case, we normally “batch” some inputs and outputs so that they can be processed in parallel. This is named as batch or mini-batch approaches.

Building on these two initial flaws, we can code a much more performant neural network as follows:

```
# Layer class for grouping neurons and using matrix operations
class Layer:
    def __init__(self, num_inputs, num_neurons, activation_function, activation_de-
rivative, init_method="xavier"):
```

```
        """
        Initialize a layer with a weight matrix and bias vector.
        Args:
            num_inputs (int): Number of inputs to this layer (i.e., number of neurons in the previous layer).
            num_neurons (int): Number of neurons in this layer.
            activation_function (function): Activation function (e.g., sigmoid, ReLU).
            activation_derivative (function): Derivative of the activation function.
        """
        if init_method == "xavier":
            self.weights = np.random.randn(num_inputs, num_neurons) * np.sqrt(1 / num_inputs)
        elif init_method == "he":
            self.weights = np.random.randn(num_inputs, num_neurons) * np.sqrt(2 / num_inputs)
        else:
            self.weights = np.random.randn(num_inputs, num_neurons) * 0.1

        self.biases = np.random.randn(1, num_neurons) * 0.1
        self.activation_function = activation_function
        self.activation_derivative = activation_derivative
        self.output = None
        self.inputs = None

    def forward(self, inputs):
        """
        Compute the outputs of this layer using matrix operations.
        Args:
            inputs (np.ndarray): The input matrix for the layer.
        Returns:
            np.ndarray: The activated outputs of this layer.
        """
        self.inputs = inputs
        z = np.dot(inputs, self.weights) + self.biases  # Linear combination
        self.output = self.activation_function(z)  # Apply activation
        return self.output

    def backpropagate(self, d_error, learning_rate):
        """
        Perform backpropagation for this layer using matrix operations.
        Args:
            d_error (np.ndarray): The gradient of the error with respect to this layer's output.
            learning_rate (float): Learning rate for gradient descent.
        Returns:
            np.ndarray: The gradient of the error with respect to the layer's inputs (for previous layers).
        """
        # Compute the error gradient w.r.t. the linear combination output z
        d_z = d_error * self.activation_derivative(self.output)

        # Calculate the gradient w.r.t. weights, biases, and inputs
        d_weights = np.dot(self.inputs.T, d_z)
        d_biases = np.sum(d_z, axis=0, keepdims=True)
        d_inputs = np.dot(d_z, self.weights.T)

        # Update weights and biases
        self.weights -= learning_rate * d_weights
        self.biases -= learning_rate * d_biases

        return d_inputs  # Return the error gradient for the inputs to this layer

# Neural network class using vectorized layers
class SpeedyNeuralNetwork:
    def __init__(self, layers):
        """
        Initialize the neural network.
        Args:
            layers (list): A list of `Layer` objects representing the neural network architecture.
        """
        self.layers = layers
```

```
70.
71.     def forward(self, inputs):
72.         """
73.         Perform the forward pass through the network.
74.         Args:
75.             inputs (np.ndarray): The input matrix for the network.
76.         Returns:
77.             np.ndarray: The output of the final layer.
78.         """
79.         for layer in self.layers:
80.             inputs = layer.forward(inputs)
81.         return inputs
82.
83.     def backpropagate(self, inputs, targets, learning_rate):
84.         """
85.         Perform the backpropagation pass through the network.
86.         Args:
87.             inputs (np.ndarray): The input matrix for the network.
88.             targets (np.ndarray): The target matrix for the network.
89.             learning_rate (float): The learning rate for gradient descent.
90.         """
91.         if targets.ndim == 1:
92.             targets = targets.reshape(1, -1)
93.         if inputs.ndim == 1:
94.             inputs = inputs.reshape(1, -1)
95.
96.         # Perform a forward pass to compute the network's output
97.         final_outputs = self.forward(inputs)
98.
99.         # Calculate the error gradient for the output layer
100.         d_error_d_output = -(targets - final_outputs)
101.
102.         # Backpropagate through each layer (in reverse order)
103.         d_error = d_error_d_output
104.         for layer in reversed(self.layers):
105.             d_error = layer.backpropagate(d_error, learning_rate)
106.
107. def train(network, inputs, targets, epochs=10000, learning_rate=0.1,
print_every=1000, batch_size=16, shuffle=True, plot=True):
108.     """
109.     Trains a neural network using backpropagation.
110.     Args:
111.         network (object): The neural network to be trained. Must have `backpropagate`
and `forward` methods.
112.         inputs (list or np.ndarray): The input data for training.
113.         targets (list or np.ndarray): The target outputs corresponding to the input
data.
114.         epochs (int, optional): The number of training iterations. Default is 10000.
115.         learning_rate (float, optional): The learning rate for the backpropagation
algorithm. Default is 0.1.
116.         print_every (int, optional): The interval (in epochs) at which to print the
total error. Default is 1000.
117.         batch_size (int, optional): The size of mini-batches for training. Default is
16.
118.         shuffle (bool, optional): Whether to shuffle the data before each epoch. De-
fault is True.
119.         plot (bool, optional): Whether to plot the results of the neural network af-
ter training. Default is True.
120.     Prints:
121.         The total error at specified intervals during training.
122.     """
123.     loss_history = []
124.     num_samples = len(inputs)
125.     total_error_last_epoch = 0
126.     for epoch in range(epochs):
127.         epoch_loss = 0
128.         # Shuffle data for each epoch
129.         if shuffle:
130.             indices = np.arange(num_samples)
131.             np.random.shuffle(indices)
132.             inputs, targets = inputs[indices], targets[indices]
133.
134.         # Train on mini-batches
```

```
135.         for start in range(0, num_samples, batch_size):
136.             end = start + batch_size
137.             batch_inputs = inputs[start:end]
138.             batch_targets = targets[start:end]
139.             # Train the network on the current mini-batch
140.             predicted_output = network.forward(batch_inputs)
141.             error = np.sum((batch_targets - predicted_output) ** 2)
142.             epoch_loss += error
143.             network.backpropagate(batch_inputs, batch_targets, learning_rate)
144.
145.         # Every once in a while, print the total error to see progress (we shoud see
it decrease)
146.         if epoch % print_every == 0:
147.             total_error = 0
148.             for input_data, target in zip(inputs, targets):
149.                 predicted_output = network.forward(input_data)
150.                 total_error += np.sum((target - predicted_output) ** 2)
151.             print(f"Epoch {epoch} - Total Error: {total_error:.4f}")
152.             total_error_last_epoch = total_error
153.         loss_history.append(epoch_loss / num_samples)
154.     if plot:
155.         plt.plot(loss_history)
156.         plt.xlabel("Epoch")
157.         plt.ylabel("Loss")
158.         plt.title("Training Loss")
159.         plt.show()
```

Notice how the train method has now a parameter called *batch_size* we make use of this parameter by computing *batch_size* inputs at a time!

Exercise 7: Use the SpeedyNeuralNetwork to compare with previous examples. You should see the performance has increased inmensly, going from minutes to few seconds.

6. Final classification task

Now that we have an optimized version of the neural network, we can increase the level of difficulty of the problems that this network can solve.

We can try, for example, to recognize handwritten digits! For this, we will use a dataset that has images of 8x8 pixels of hand-written digits. The dataset can be found in [16].

First, load the data, encode it using one-hot encoding, and split the training and test data:

```
 1. from sklearn.datasets import load_digits
 2. digits = load_digits()
 3. inputs = digits.data
 4. targets = digits.target
 5.
 6. # One hot encode the targets
 7. num_classes = len(np.unique(targets))
 8. targets = np.eye(num_classes)[targets]
 9.
10. # Split the data into training and testing sets
11. inputs_train, inputs_test, targets_train, targets_test = train_test_split(inputs, tar-
gets, test_size=0.2, random_state=42)
```

Then we will define the network, in which the input is going to be 64. This is due to the images in the dataset being 8x8 pixels, so 64 values for a given number.

```
1. layers = [
2.     Layer(64, 32, sigmoid, sigmoid_derivative),
3.     Layer(32, 10, sigmoid, sigmoid_derivative),
4. ]
5. network = SpeedyNeuralNetwork(layers)
```

```
6. train(network, inputs_train, targets_train, epochs=3000, learning_rate=0.01,
print_every=250)
```

After little time, our network will be trained and fully capable for recognizing hand-drawn digits!:

```
 1. # Evaluate the network on the test set
 2. correct = 0
 3. for input_data, target in zip(inputs_test, targets_test):
 4.     predicted_output = network.forward(input_data)
 5.     predicted_digit = np.argmax(predicted_output)
 6.     true_digit = np.argmax(target)
 7.     if predicted_digit == true_digit:
 8.         correct += 1
 9.
10. accuracy = correct / len(inputs_test) * 100
11. print(f"Accuracy on test set: {accuracy:.2f}%")
12.
13. #plot some digits and the predicted output
14. plt.figure(figsize=(12, 6))
15. for i in range(10):
16.     plt.subplot(2, 5, i + 1)
17.     plt.imshow(inputs_test[i].reshape(8, 8), cmap="gray")
18.     predicted_output = network.forward(inputs_test[i])
19.     predicted_digit = np.argmax(predicted_output)
20.     true_digit = np.argmax(targets_test[i])
21.     plt.title(f"Predicted: {predicted_digit}\nTrue: {true_digit}")
22.     plt.axis("off")
23. plt.tight_layout()
24. plt.show()
```

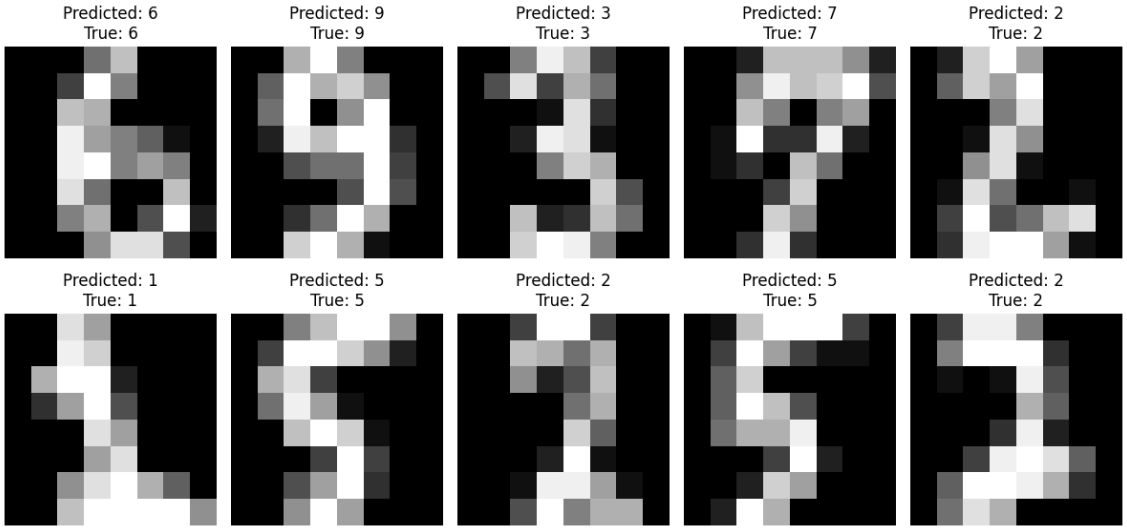


Figure 8. Resulting classification of hand-written digits.

References

1. Google Colab, available in URL: https://colab.research.google.com/.
2. NumPy scientific computing in Python, available in URL: https://numpy.org/doc/stable/index.html
3. Matplotlib, plotting library for Python, available in URL: https://matplotlib.org/
4. Scikit-learn, machine learning in Python, available in URL: https://scikit-learn.org/stable/
5. Dot operation from NumPy, available in URL: https://numpy.org/doc/stable/reference/generated/numpy.dot.html
6. Python list comprehensions, available in URL: https://docs.python.org/3/tutorial/datastructures.html#list-comprehensions
7. Setting the learning rate, Jeremy Jordan, available in: https://www.jeremyjordan.me/nn-learning-rate/
8. California Housing dataset, available in: https://www.dcc.fc.up.pt/~ltorgo/Regression/cal_housing.html
9. Iris dataset, available in: https://archive.ics.uci.edu/dataset/53/iris
10. Xavier initialization, available in: https://proceedings.mlr.press/v9/glorot10a.html
11. He initialization, available in: https://arxiv.org/abs/1502.01852
12. Lecun initialization, available in: https://yann.lecun.com/exdb/publis/pdf/lecun-98b.pdf
13. Momentum in gradient descent, Scaler, available in: https://www.scaler.com/topics/momentum-based-gradient-descent/
14. Vanishing gradient, Wikipedia, available in: https://en.wikipedia.org/wiki/Vanishing_gradient_problem
15. Underfitting and overfitting, Ryan Holbrook, Kaggle, available in: https://www.kaggle.com/code/ryanholbrook/overfitting-and-underfitting
16. Handwritten digits dataset, available in: https://archive.ics.uci.edu/dataset/80/optical+recognition+of+handwritten+digits.