Deep Learning (AIC-4201)

Final Project (Digit recognition)

Students:

Bruno CASTRO

Matěj SMID

Professor:

Giovanni CHIERCHIA

January 15, 2022

The code develop during this assignment is available in GitHub

Introduction

In this project developed for the Deep Learning class at ESIEE Paris, we were challenged to put into practice the concepts and theories we saw in the classroom. For this, we had in our disposal a Python notebook. Following the steps in the document, we developed the entirety of this project.

In the chapters below, we detail the implementation and concepts behind each quiz in the Python notebook. Each one of them is an important step in the implementation of a neural network. In the end, we had a fully functional network, capable of classifying the proposed problem (digit recognition).

Quiz 1 (Softmax):

Softmax is a common activation function. In our network, It'll be mainly used as the activation function for the final layer (as it is needed for such classification problems). The implementation is as follows:

```
def softmax(z):
    """Softmax of the rows of z"""
    # Compute the exponential
    z_exp = np.exp(z)
    print(f"z_exp.shape = {z_exp.shape}")
    # Sum the elements row-by-row. Don't forget to keep the dimensions!
    z_sum = np.sum(z_exp, axis=1)
    print(f"z_sum.shape = {z_sum.shape}")
    # Compute the division. It should automatically use broadcasting!
    s = z_exp / z_sum[:,None]
    print(f"s.shape = {s.shape}")
    # Return the result
    return s
```

The softmax is calculated by the following formula:

$$\text{softmax}(Z) = \begin{bmatrix} \vdots & & \vdots \\ \frac{\exp(Z_{n,1})}{\exp(Z_{n,1}) + \cdots + \exp(Z_{n,N_{\text{units}}})} & \cdots & \frac{\exp(Z_{n,N_{\text{units}}})}{\exp(Z_{n,1}) + \cdots + \exp(Z_{n,N_{\text{units}}})} \\ \vdots & & \vdots \end{bmatrix}$$

Quiz 2 (outputs from inputs and params):

This implementation is really straightforward: the output of a dense layer will be equal to the inputs multiplicated by the weights, added to the bias in the end. The overall formula is the following:

$$Z = XW + \text{b}$$
$$A = \text{activation}(Z)$$

Our implementation is as following:

```
def dense_layer(inputs, weights, bias, activation=None):
  """
  Arguments:
  inputs  -- input matrix of shape (n_samples, n_input)
  weights -- weight matrix of shape (n_input, n_output)
  bias    -- bias vector of shape (1, n_output)
  activation -- name of the nonlinear function

  Returns:
  outputs -- output matrix of shape (n_samples, n_output)
  """

  # Step 1: linear transform
  outputs = inputs @ weights + bias
  # Step 2: nonlinear activation
  if activation == 'softmax':
      outputs = softmax(outputs)
  elif activation == 'relu':
      outputs = np.maximum(np.zeros_like(outputs), outputs)
```

```
return outputs
```

The operator '@' takes care of the matrix multiplication. After calculating the output, it goes through an activation function before going into the next layer. In this part of the implementation, we have two activations: softmax (implemented earlier) and relu (implement inline).

Quiz 3 (random initialization):

In this part of the implementation, we needed to implement a way to initialize our parameters. In a neural network, this is a very important step, as the initialization of these can impact profoundly in the performance and the speed in which the network learns,

A random initialization is the best choice in this scenario. We initialized all biases with zeros, and all weights following the following distribution (as advised by the literature).

$$\sqrt{\frac{2}{\text{dimension of the previous layer}}}$$

Our implementation is the following:

```
def initialize_parameters(n_input, n_hidden, n_output):
  """
  Arguments:
  n_input  -- Size of network input
  n_hidden -- List of sizes of hidden layers
  n_output -- Size of network output (2nd layer)

  Returns:
  parameters -- collection of weight matrices and bias vectors
  """

  W1 = np.random.randn(n_input, n_hidden) * np.sqrt(2./n_input)
  b1 = np.zeros((1, n_hidden))
  W2 = np.random.randn(n_hidden, n_output) * np.sqrt(2./n_hidden)
```

```
    b2 = np.zeros((1, n_output))

    return W1, b1, W2, b2
```

Quiz 4 (output of a two layer NN):

In this part of the implementation, we needed to compute the output of our two layer neural network. For this, we need to use the functions implemented before, using the correct parameters.

The implementation is as following:

```
def twolayer_network(X, parameters):
    """
    Arguments:
    X -- input matrix of shape (n_samples, n_input)
    parameters -- W1, b1, W2, b2

    Returns:
    Y -- output matrix of shape (n_samples, n_output)
    """

    # Parameters
    W1, b1, W2, b2 = parameters

    # For backwards compatibility
    W1 = W1
    b1 = b1
    # Forward propagation: INPUT -> LAYER 1 -> LAYER 2 -> OUTPUT
    A0 = X
    A1 = dense_layer(A0, W1, b1, 'relu')
    A2 = dense_layer(A1, W2, b2, 'softmax')

    return A2
```

In this implementation, we receive the parameters and inputs as input of the function. Then we call the function "dense_layer()" twice: one time for the first layer, with the 'relu' activation function, and a second time for the output layer, with the 'softmax' activation function.

Quiz 5 (one-hot vectors):

For a classification problem (such as the one we're working on), using a scalar value as the output of the network can be troublesome, and impact the learning (as the learning process may think there's a closer relation between two consecutive integers 'n' and 'n+1' then two far apart from each other, what is not true).

To avoid the problem, we can convert our output from an integer to a vector of size max(outputs), where all values ares 0, except the position it was classified in. The explanation goes as follows:

$$\mathbf{y}_n = [0 \ \dots \ 0 \ \underbrace{1}_{\text{class } c_n} \ 0 \ \dots \ 0]^\top \in \mathbb{R}^{N_{\text{output}}}.$$

```
labels[0] = 1  -->  one_hot[0] = [0, 1, 0, 0, 0, 0, 0]
labels[1] = 2  -->  one_hot[1] = [0, 0, 1, 0, 0, 0, 0]
labels[2] = 6  -->  one_hot[2] = [0, 0, 0, 0, 0, 0, 1]
labels[3] = 4  -->  one_hot[3] = [0, 0, 0, 0, 1, 0, 0]
```

Our implementation is the following:

```
def to_categorical(labels):
  """
  Convert integers into one-hot vectors.

  Arguments:
  labels - vector of integers from 0 to C-1

  Returns:
  one_hot - matrix of shape (labels.size, C) with a one-hot encoded row for each element of 'labels'
  """
```

```
    # Number of elements in 'labels'
    size = labels.shape[0]
    # One plus the max value in 'labels'
    C = np.max(labels) + 1

    # Create a zero matrix with the right shape
    one_hot = np.zeros((size, C))

    # Set the value 1 in the right positions
    idx1 = np.arange(size)
    idx2 = labels

    one_hot[idx1, idx2] = 1

    return one_hot
```

Quiz 6 (cross entropy):

Our network needs a way of knowing if it is performing well or not, so the learning process can occur. For this, we need some function to compare network outputs to the expected outputs in the training data, and rank them in some way. One such mathematical tool is the cross entropy.

The cross entropy works as following:

$$\text{cross-entropy}\left(\hat{Y}, Y_{\text{true}}\right) = \frac{1}{N_{\text{samples}}} \sum_{n=1}^{N_{\text{samples}}} -\mathbf{y}_n^{\top} \log\left(\hat{\mathbf{y}}_n\right).$$

And our implementation is below:

```
def cross_entropy(Y, Y_true):
    """
    Arguments:
    Y -- network's output matrix of shape (n_samples, n_output)
    Y_true -- true target matrix of shape (n_samples, n_output)
```

```
    Returns:
    J -- scalar measuring the cross-entropy between Y and Y_true
    """

    J = - np.sum(Y_true * np.log(Y)) / Y.shape[0]

    return J
```

Quiz 7 (training loss):

For the training loss part of the implementation, we just need to use the loss function implemented earlier (cross entropy) on top of the outputs of our twolayer NN, also implemented earlier. Our implementation is as follows:

```
def training_loss(parameters, X, Y):
    """
    Arguments:
    X -  input matrix of shape (n_samples, n_input)
    Y - output matrix of shape (n_samples, n_output)

    Returns:
    cost - cross entropy between the network predictions and the true outputs
    """

    # Run the network on the inputs to obtain the predictions

    #Y_pred = twolayer_network(X, parameters)
    Y_pred = twolayer_network(X, parameters)
    # Compute the cross entropy between the predictions and the true outputs
    cost = cross_entropy(Y_pred, Y)

    return cost
```

Quiz 8 (run the network):

In this part of the implementation, we just need to predict the outputs in our test set, using the trained network. For this, we send the trained parameters to our twolayer_network along with the test_set, and run the argmax function on the output, which gives us the predicted class. The implementation is the following:

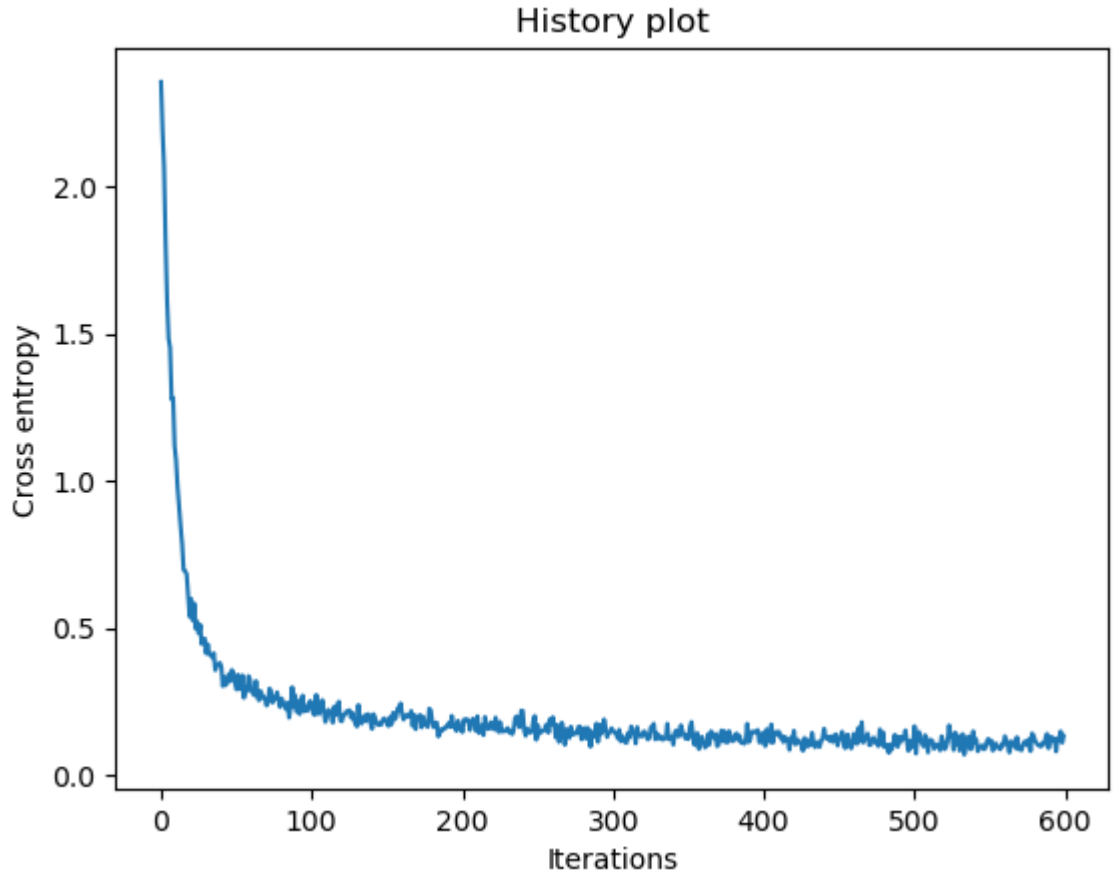
```
def prediction(X, params):
  """
  Arguments:
  X -- input matrix of shape (n_samples, n_input)
  params -- W1, b1, W2, b2

  Returns:
  c -- prediction for each row of the input matrix
  """

  # Run the network on the inputs
  Y = twolayer_network(X, params)
  # Compute the argmax of each probability vector
  c = np.argmax(Y, axis=1)

  return c
```

Quiz 9 (Stochastic optimization):

To train more efficiently, we implement a stochastic optimization scheme. This involves splitting the input-output pairs into B batches, and then computing the loss function, running the GD and updating the weights for each batch separately.

```
def __call__(self, params, iteration):

        # extract the current batch
        X, Y = self.get_batch(iteration)
        # run the network on the batch
        #Y_pred = twolayer_network(X, params)
        Y_pred = multilayer_network(X, params,
dropout_rate=self.dropout)

        # compute the cross entropy
        cost = cross_entropy(Y_pred, Y)

        # save it for later
        self.history.append(cost if isinstance(cost, float) else
cost._value)

        return cost
```

Because this algorithm computes the gradient only for a subset of the training data, the loss function is not strictly decreasing, but instead it fluctuates.



However, this approach is not Stochastic Gradient Descent, which computes the gradient only from one sample instead of batches.

Quiz 10 (process an image with digits):

In the step of the implementation, we needed to process the images of a real image, and reshape them accordingly, so we could run them through our network.

For this task, we used the cv2 library, which has implemented functionality to work with and process images. The actual implementation is out of the scope of this project, but the function "find_digits()" returns us the locations of recognised digits in an image. To make these digits proper to work with our network, we just need to reshape them. The implementation is as follows:

```
import cv2
```

```
from lab04_data.digits import find_digits

def digit_recognition(image_name):

  # Digit extraction
  image = cv2.imread(image_name)
  digits, rects = find_digits(image)

  # Preprocessing: reshape and normalize the digits
  digits = digits.reshape(-1,28*28)/255

  # Network prediction: classify the digits
  labels = multilayer_network(digits, trained_params).argmax(axis=1)

  # Visualization
  plt.figure(figsize=(8,8))
  for tag, rect in zip(labels, rects):
      cv2.rectangle(image, (rect[0], rect[1]), (rect[0] + rect[2], rect[1] + rect[3]), (0, 255, 0), 3)
      cv2.putText(image, str(tag), (rect[0], rect[1]-5), cv2.FONT_HERSHEY_DUPLEX, 1.5, (255, 0, 0), 3)
  plt.imshow(image)
  plt.show()
```

Quiz 11 (Multilayer network):

For this quiz, we had to change the internal structure of most classes and functions. We changed the function *initialize_parameters()* take 3 params: int, list and int; instead of the int, int, int. The second argument is a list of integers indicating the number of hidden neurons in each hidden layer. The function returns a tuple of weights and biases.

```
def initialize_parameters(n_input, n_hidden:list, n_output):
    """
    Arguments:
    n_input  -- Size of network input
    n_hidden -- List of sizes of hidden layers
    n_output -- Size of network output (2nd layer)

    Returns:
    parameters -- collection of weight matrices and bias vectors Whidden,
bhidden, Wout, bout
    """
```

```
    # For backwards compatibility

    if type(n_hidden) is not list:
        n_hidden = [n_hidden]

    n = [n_input] + n_hidden + [n_output]

    W = [None]*(len(n)-1)
    b = [None]*(len(n)-1)
    for i in range(len(n)-1):
        W[i] = np.random.randn(n[i], n[i+1]) * np.sqrt(2./n[i])
        b[i] = np.zeros((1, n[i+1]))

    params = tuple()
    for i in range(len(n)-1):
        params += (W[i], b[i])

    return params
```

Next, we added a *multilayer_network()* function. Here we loop over all the weight matrices in the tuple of parameters and apply a dense layer transformation on each. We are able to pass a list of activation functions that will be used. We also use dropout, we pass a parameter *dropout_rate* to the function and then we use that to zero out different neuronal connections as a way to regularize the network.

```
def multilayer_network(X, parameters, activations:list=None,
dropout_rate:float=0.):
    """
    Arguments:
    X -- input matrix of shape (n_samples, n_input)
    parameters -- W1, b1, W2, b2

    Returns:
    Y -- output matrix of shape (n_samples, n_output)
    """
    p = 1 - dropout_rate

    # Parameters
    W = parameters[::2]
    b = parameters[1::2]

    if activations is None:
        activations = ['relu'] * (len(W)-1) + ['softmax']
```

```
    for i in range(len(W)):
        X = dense_layer(X, W[i], b[i], activations[i])

        # Dropout random generator messes with the assert tests so I had
to ...
        X *= (np.random.rand(*X.shape) < p)/p


    return X
```

We modified the *StochasticOptimizationProblem* and the *stochastic_learning()* function to use *multilayer_network()* instead of *twolayer_network()*.

We also added some additional activation functions to the *dense_layer()* function.

```
def dense_layer(inputs, weights, bias, activation=None):
    """
    Arguments:
    inputs  -- input matrix of shape (n_samples, n_input)
    weights -- weight matrix of shape (n_input, n_output)
    bias    -- bias vector of shape (1, n_output)
    activation -- name of the nonlinear function

    Returns:
    outputs -- output matrix of shape (n_samples, n_output)
    """


    # Step 1: linear transform
    outputs = inputs @ weights + bias
    # Step 2: nonlinear activation
    if activation == 'softmax':
        outputs = softmax(outputs)
    elif activation == 'relu':
        outputs = np.maximum(np.zeros_like(outputs), outputs)
    elif activation == 'tanh':
        outputs = np.tanh(outputs)
    elif activation == 'sigmoid':
        outputs = 1 / (1 + np.exp(-outputs))
```

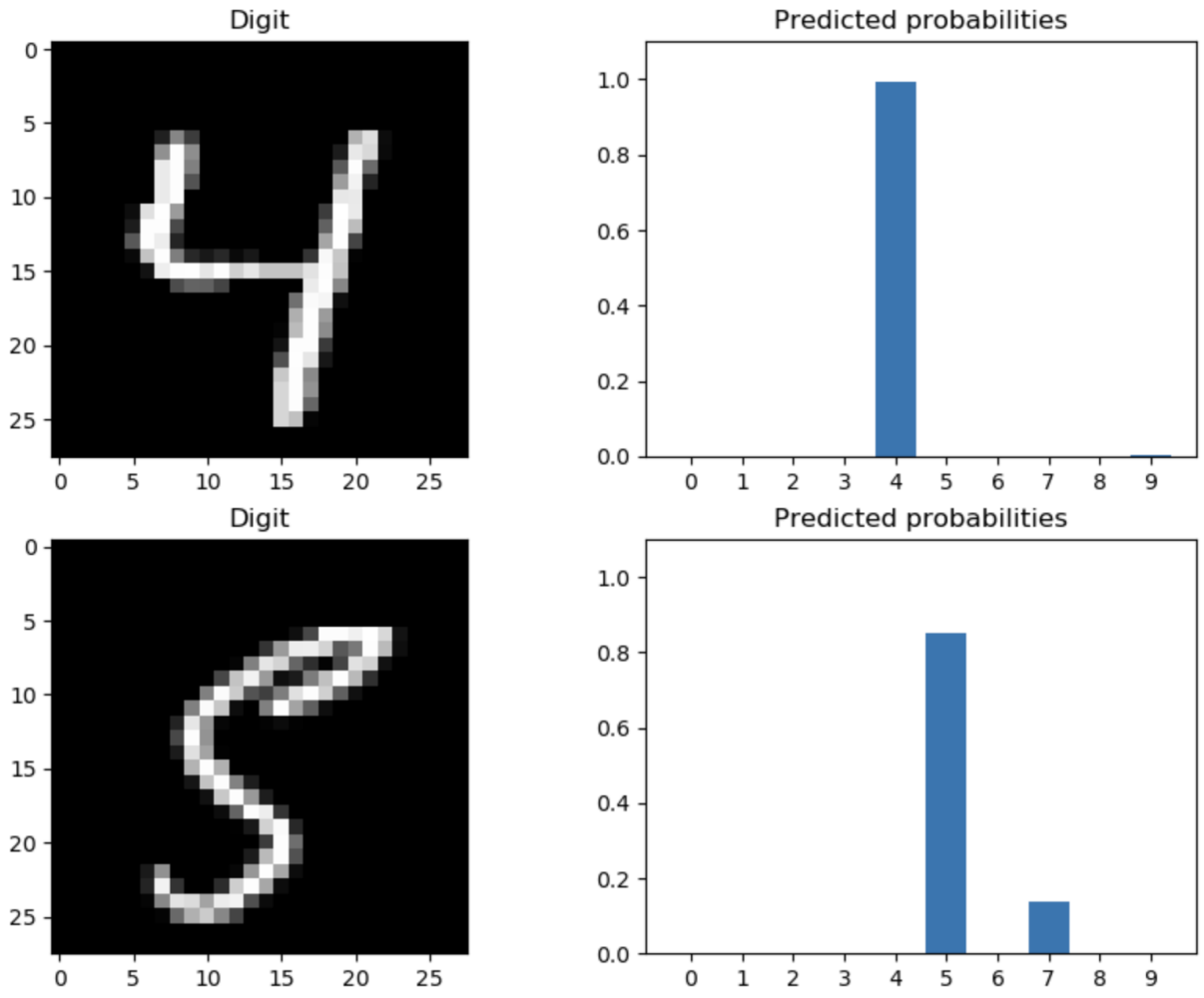
```
    elif activation == 'leaky_relu':
        outputs = np.maximum(0.01 * outputs, outputs)
    elif activation == 'elu':
        outputs = np.where(outputs > 0, outputs, np.exp(outputs) - 1)
    elif activation == 'selu':
        _lambda = 1.0507
        outputs = np.where(outputs > 0, _lambda * outputs, _lambda *
np.exp(outputs) - _lambda)
    elif activation == 'linear' or activation is None:
        pass

    return outputs
```

Conclusion:

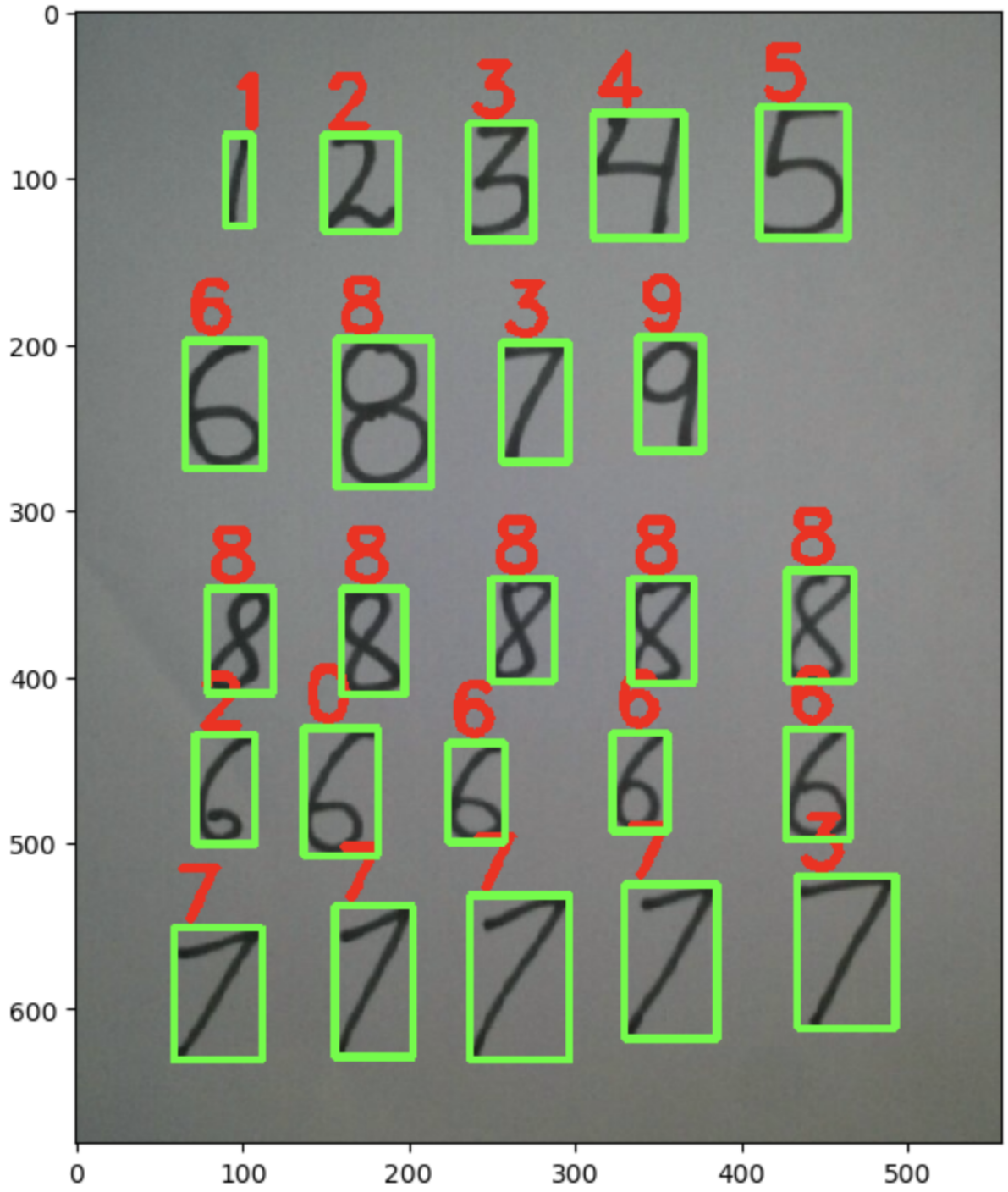
After all the Quizzes were done and all implementations finalized, we were able to obtain a fully functional neural network, with surprising performance.

Overall, even though the digit classification problem is a relatively simple one to tackle in the neural network realm (and a quite famous one), we were still impressed by the results obtained and surprised by pulling it off with such ease.

Below, we attached some of our results:



Digit classification



Processed image classification