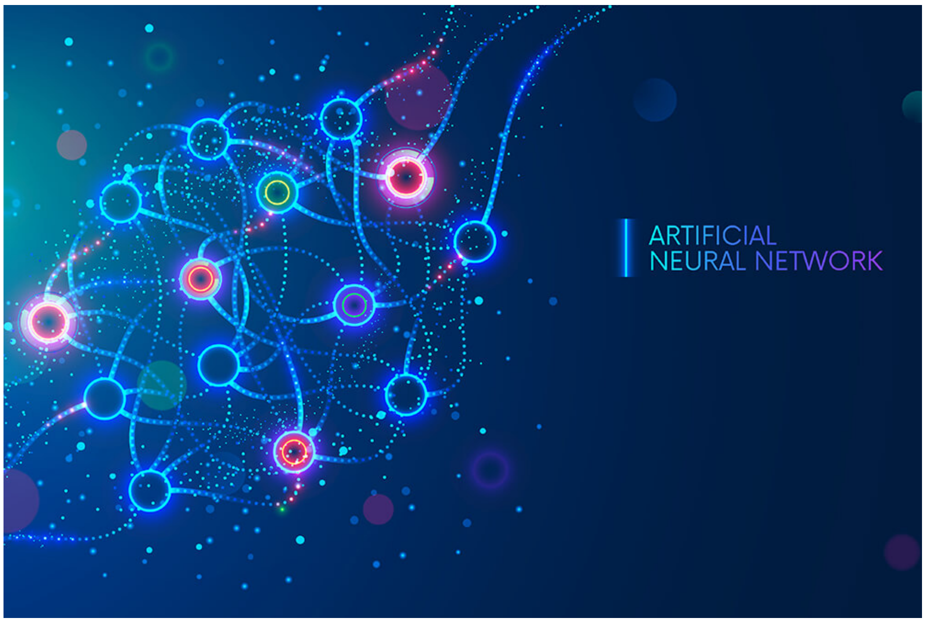
Sébastien BOIS - Maxime BOUET - AIC

Project report

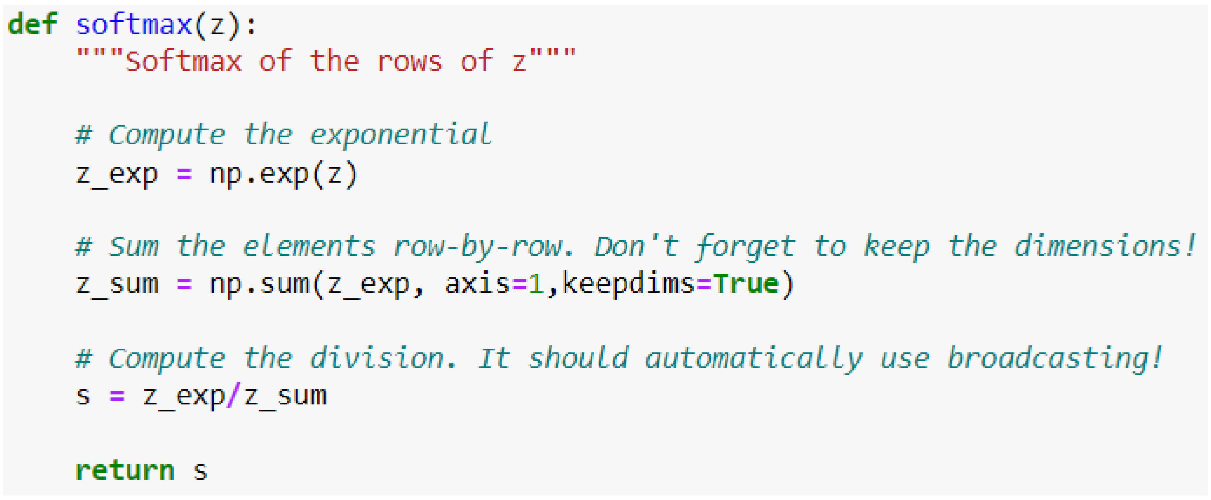


Quiz 1:

In this quiz, we want to create the softmax function. This function is generally used as an activation method for a neuron in a neural network. It is defined as:

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

We often use this activation function for classification tasks since the sums of all the elements of each column should be equal to 1.



```python
def softmax(z):
    """Softmax of the rows of z"""

    # Compute the exponential
    z_exp = np.exp(z)

    # Sum the elements row-by-row. Don't forget to keep the dimensions!
    z_sum = np.sum(z_exp, axis=1,keepdims=True)

    # Compute the division. It should automatically use broadcasting!
    s = z_exp/z_sum

    return s
```

Quiz 2:

```python
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(0,outputs)

    return outputs
```

The dense layer is the most common layer in a neural network. It only computes the linear transformation $Y = W.X + B$ and apply a non-linear activation function to this result. Here, we only consider two activation functions: softmax and ReLU. Y is the linear output, W the weight matrix, X the input matrix and B the bias vector.

N.B: The ReLU function returns 0 if the input is negative and the input itself if it is positive.

Quiz 3:

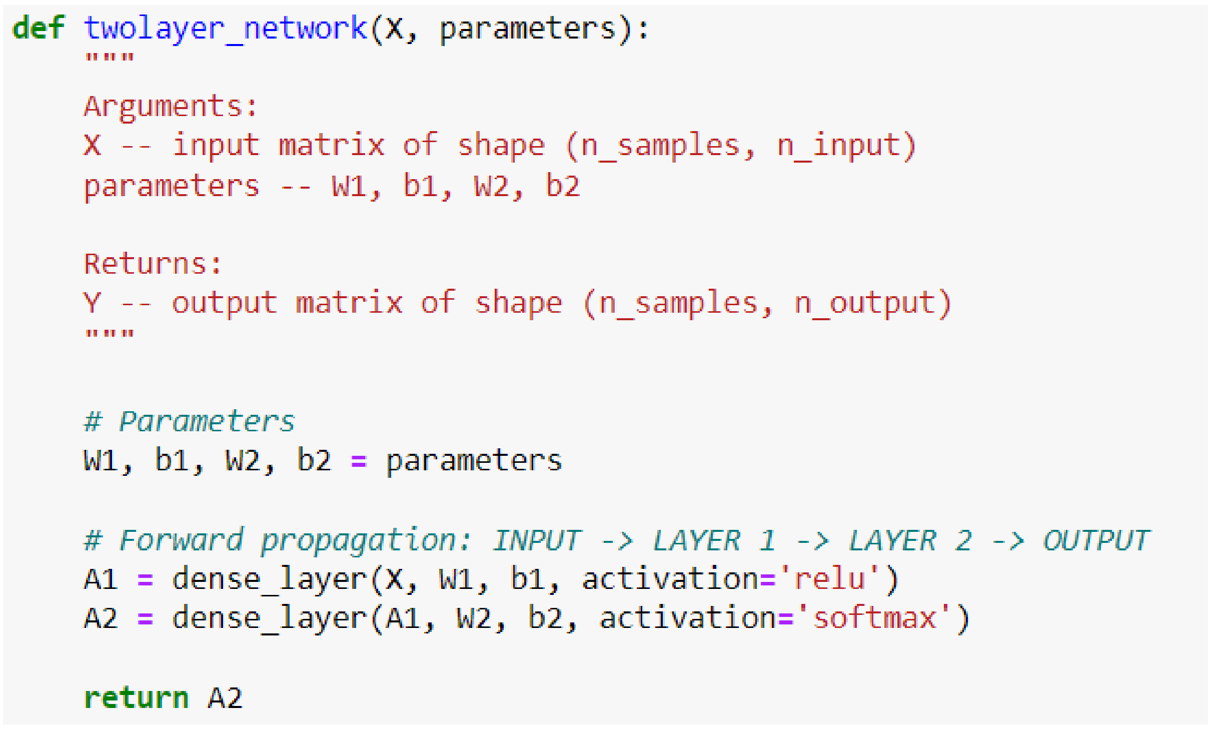
```python
def initialize_parameters(n_input, n_hidden, n_output):
    """
    Arguments:
    n_input  -- Size of network input
    n_hidden -- Size of hidden layer (1st layer)
    n_output -- Size of network output (2nd layer)

    Returns:
    parameters -- array of arrays W1, b1, W2, b2
    """

    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
```

To train a model, it is important to initialize the base parameters. In neural networks, this initialization has an important effect on the results. Indeed, initializing the weights matrix to a uniform matrix would lead each neuron to learn in the same way. Obviously, the results would be wrong. Affecting random values to those matrices would be a good solution. However, it is import to normalize the values with a scaling factor of $\sqrt{\frac{2}{previous\ dimension}}$ otherwise we would have some problems because of the potential differences between weights. Since we use a matrix multiplication in the dense layer, we must respect the norms of size of the matrix. To be able to perform a matrix multiplication, we must have $X.Y\ =\ (n,\ m).(m,\ p)$ with the couples (n, m) and (m, p) the dimensions of the matrix X and Y.

Concerning the bias vectors, we can initialize it to zero without having any bad impact on the results. The dimension must only be $(1,\ p)$.

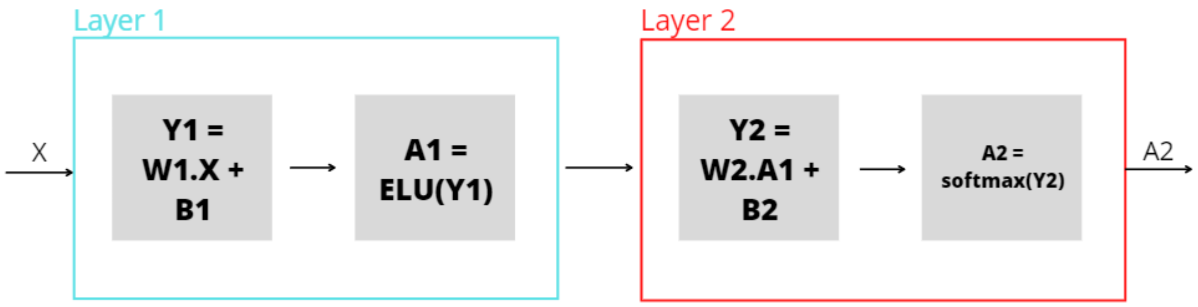
Lastly, since we define a two-layer network, we must have two weight matrices and two bias vectors, one for each layer.

Quiz 4:



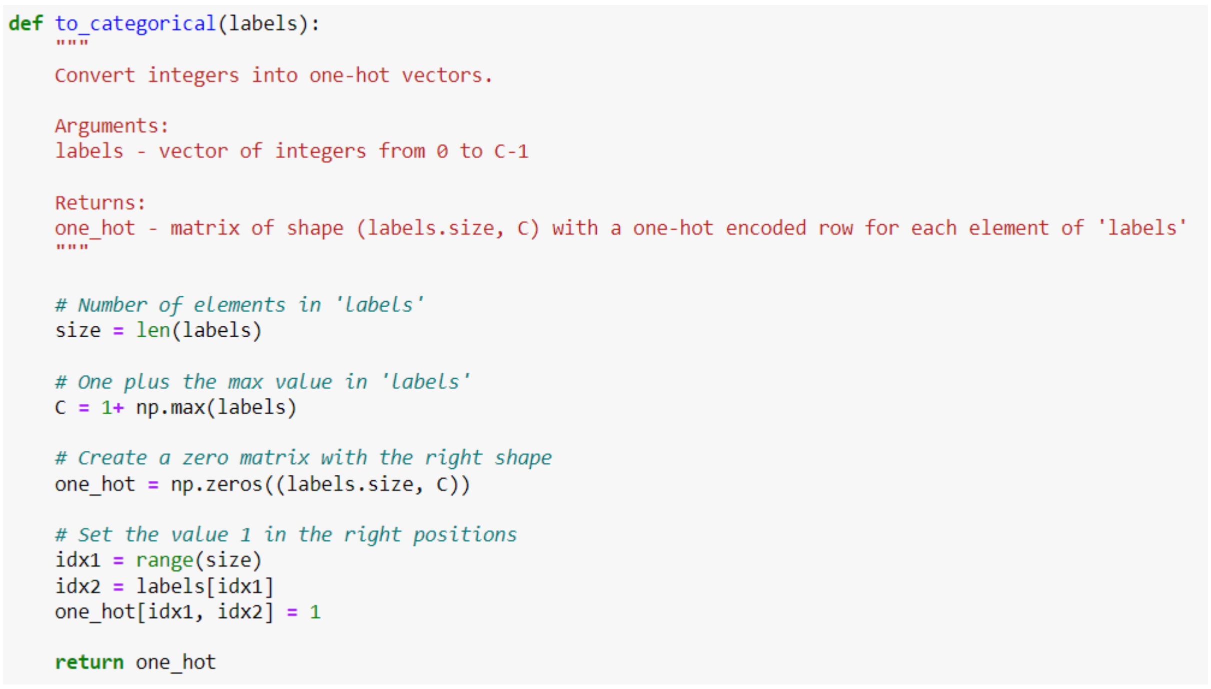
```python
def twolayer_network(X, parameters):
    """
    Arguments:
    X -- input matrix of shape (n_samples, n_input)
    parameters -- W1, b1, W2, b2

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

    # Parameters
    W1, b1, W2, b2 = parameters

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

    return A2
```

A two layer neural network is defined with the following pattern:



Then, we must call the initialize_parameters function to obtain the weight and bias matrices. After that, we can create each dense layer of the neural network and compute their results. In classification problems, we often use a sigmoid activation function because it gives positive results. We will see that we must have positive results to compute the cross-entropy function and to be able to evaluate our network. Here, we choose the softmax function.

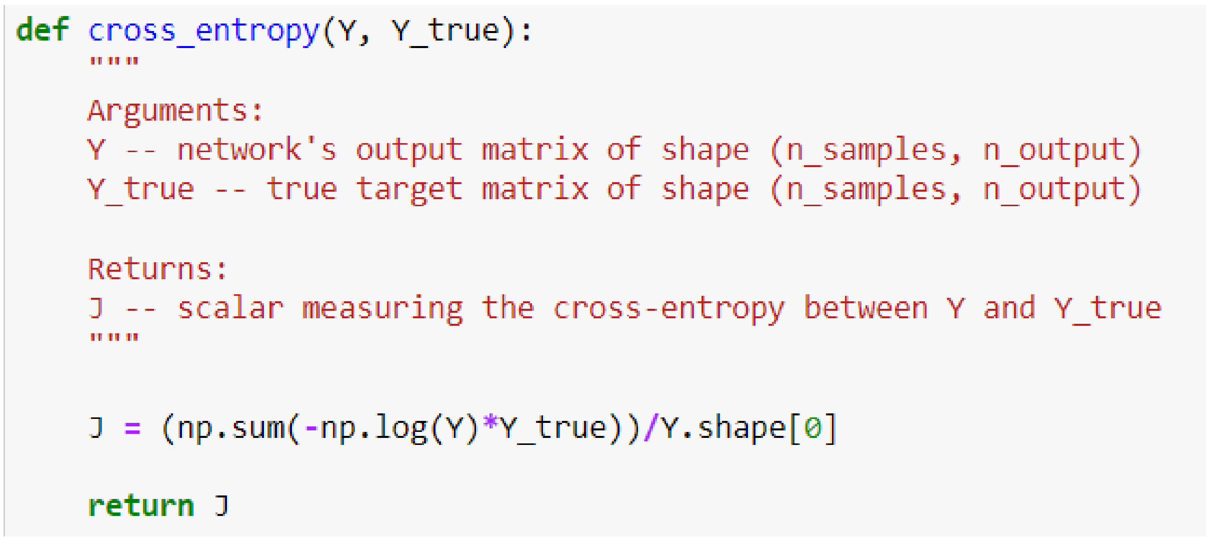
Quiz 5:



```
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 = len(labels)

    # One plus the max value in 'labels'
    C = 1+ np.max(labels)

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

    # Set the value 1 in the right positions
    idx1 = range(size)
    idx2 = labels[idx1]
    one_hot[idx1, idx2] = 1

    return one_hot
```

The objective of this function is to transform an array of integers into a matrix of one hot vector representing each integer by an array of zero and a one to the represented index. For example, if we want to encode 3 as a one-hot vector, it would result in an array like $[0\ 0\ 0\ 1\ 0\ 0 \ldots\ 0]$. We use this representation to be able to transform a numerical result into a pure classification. In our problems, we will literally encode the labels as one-hot vectors to compare it to our network's result.

Quiz 6:



```python
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(-np.log(Y)*Y_true))/Y.shape[0]

    return J
```

For this part, we just had to follow the formula given in the statement, which corresponds to this one:

$$H(p,q) = -\sum_{x\in\mathcal{X}} p(x)\,\log q(x) \quad \textbf{(Eq.1)}$$

So in our case, we have $p \ = \ Y \ and \ q \ = \ Y\_true$. At the end, we divide by the number of samples which is $Y.shape[0]$ because we will minimize this cost function to obtain the best parameters for our network and having huge gaps between values would lead to a significant loss of speed.

Quiz 7:

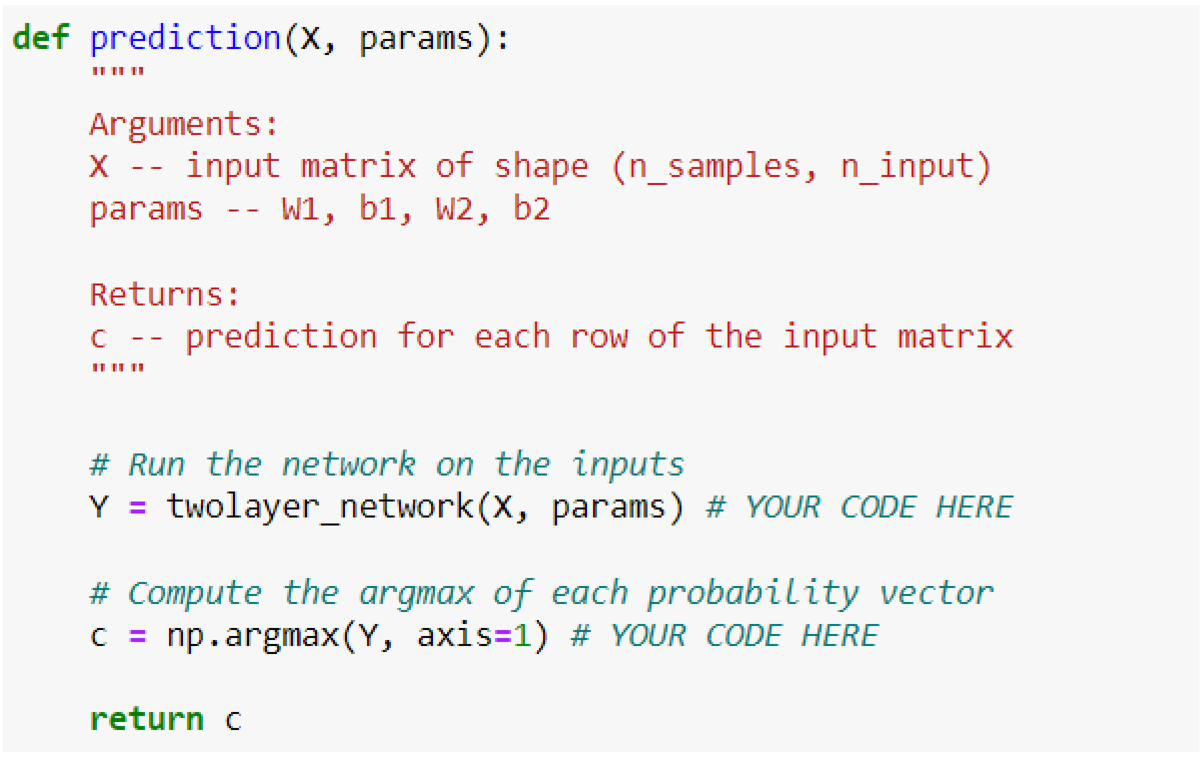
```python
def training_loss(parameters, X, Y):
    """
    Arguments:
    X -  input matrix of shape (n_samples, n_input)
    Y - output matrix of shape (n_samples, n_output)

    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)

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

    return cost
```

Then, here, the goal was to define the cost function of our problem such that:

$$J(\theta) = \frac{1}{N_{\text{samples}}} \sum_{n=1}^{N_{\text{samples}}} -\mathbf{y}_n^{\mathsf{T}} \log \left(f_\theta(\mathbf{x}_n) \right).$$

We are only going to use previously coded functions. So first, we have to run the network with the specified inputs to obtain the predictions using the previous twolayer_network function. Next, we call the cross_entropy function to compute the cross entropy between the predictions and the real value. This result corresponds to the cost of our neural network that we'll try to minimize to obtain the optimal parameters.

Quiz 8:



```python
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) # YOUR CODE HERE

    # Compute the argmax of each probability vector
    c = np.argmax(Y, axis=1) # YOUR CODE HERE

    return c
```

For this function, we want to calculate the most likely class prediction for each of the inputs by following this formula:

$$\text{prediction}(\mathrm{x}) = \arg\max f_{\theta}(\mathrm{x})$$

We first run the network with the call of the twolayer_network function. After, we have to compute the argmax of each probability vector but the network output is a matrix so we just select the appropriate axis which is 1. Then, we obtain a vector with the most voted classes which will correspond to the final prediction.

Quiz 9:

```
#----- BEGIN QUIZ -----#

def setup(self):
    self.batches = []
    for n in range(self.num_batches):
        # index slice for the n-th batch
        idx = slice(self.batch_size*n, self.batch_size * (n+1))
        self.batches.append(idx)


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)

    # 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

#----- END QUIZ -----#
```

In this part, the final goal was to define a stochastic version of the cost function. The point of using a stochastic gradient descent is to avoid the saddle points and the plateaus. Also, it computes faster than a classic gradient descent. The reason why its performance is better is because it is an approximation of the classic gradient descent. Indeed, it computes on batches of data whereas the classic one computes on all the dataset.

Then we had to define a way to create the batches of data by slicing the original dataset into small parts.

The first step was to define in the function setup() the array containing the starting indexes for each batch:

```
indices_of_1st_batch = slice(0, self.batch_size)
indices_of_2nd_batch = slice(self.batch_size, 2 * self.batch_size)
...
```

So we generalized it with this formula:

```
idx = slice(self.batch_size*n, self.batch_size * (n+1))
self.batches.append(idx)
```

Then, in the __call__ function, it's pretty fast, we just had to call the previously written twolayer_network() and cross_entropy() functions to compute the cost and we passed it to the history array.

The difficulty of this quiz was on the generalization of the index to make a changing number of slices.

With those methods, it is now possible to make a gradient descent by initializing the object that we just created with the required parameters, computing the gradient of the cost and optimizing the default parameters with the ADAM function from the Keras library.

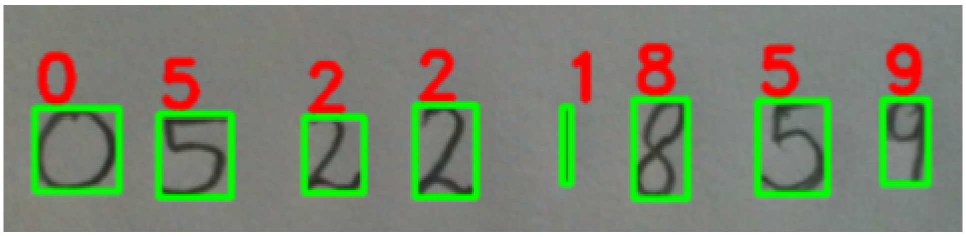
Quiz 10:

```
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(digits.shape[0], 28 * 28) / 255 # YOUR CODE HERE

    # Network prediction: classify the digits
    labels = prediction(digits, trained_params) # YOUR CODE HERE

    # 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]
        cv2.putText(image, str(tag), (rect[0], rect[1]-5), cv2.FONT_HERSHEY_D
    plt.imshow(image)
    plt.show()
```

For this last quiz, we want to preprocess a list of digit images, and run the network on them to predict their classes.
The first step was to preprocess, so reshape and normalize the digits.
We therefore use the reshape() function which takes as its first parameter the matrix that we want to reshape and the desired dimensions. Since we want to flatten the image dimensions, we reshape the array from the size (dim0, 28, 28) to the shape (dim0, 28*28).
After that, we divide everything by 255 in order to normalize.

Next, for the labels, we just have to call the prediction function in order to classify the digits.

We obtain the following results:



The model works pretty well since every prediction fits the handwritten number.

Multilayer neural networks:

The goal of this quiz is to modify the functions defined for the two-layer network to implement multi-layer networks. It would be useless to implement a network with more layers without defining more activation functions. Then, we implemented the hyperbolic tangent function, the leaky ReLU, the SeLU and ELU functions.

Activation functions:

- hyperbolic tangent:

This function follows the formula: $tanh(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}}$. It is often used as an alternative for the sigmoid functions. Indeed, the derivative of the hyperbolic tangent function looks a lot like the sigmoid function. Its results are taken between -1 and 1.

- Leaky ReLU:

The Leaky Rectified Linear Activation follows the formula:

$$f(x) = \begin{cases} x, & \text{if } x \geq 0 \\ ax, & \text{if } x < 0 \end{cases}.$$

It is a derivative of the ReLU function which returns either zero or the input itself depending on the sign of the input. The Leaky ReLU function follows the same scheme but when the input is negative, it returns a factor of this input instead of zero.

- SeLU

Scaled Exponential Linear Units follows the formula:

$$f(x) = \lambda x \qquad \text{if } x > 0$$

$$f(x) = \lambda\alpha(e^x - 1) \qquad \text{if } x \leq 0$$

Again, it is a derivative of the ReLU function which implements self-normalization. Its results' mean and variance automatically converge to zero. It helps to avoid vanishing gradient issues and it computes faster than the classic ReLU.

- ELU

Exponential Linear Unit function follows the formula:

$$y = ELU(x) = \exp(x) - 1 \text{ ; if } x<0$$

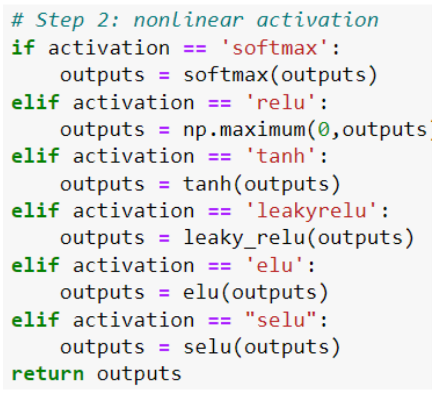
$$y = ELU(x) = x \text{ ; if } x \geq 0$$

As above, it is a derivative of the ReLU function which avoids both vanishing and exploding gradients. It computes faster than the ReLU function and it also prevents dying neurons.

Layers implementation:

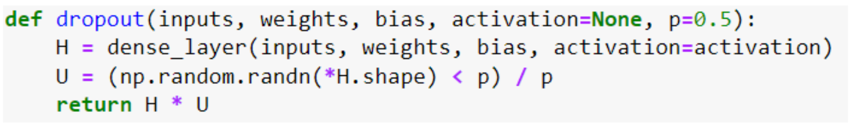
For this task, we had to take into account two types of layers: the dense layer, which is the same as for the two-layer network adapted to multi-layers, and the dropout layer. The dropout layer helps to prevent overfitting by randomly putting some training values to zero. The dropout rate is fixed arbitrarily between 0 and 1. The bigger this variable is, the lower the dropout rate is.

The dense layer is exactly the same as before except that we implemented the new activation functions.



```python
# Step 2: nonlinear activation
if activation == 'softmax':
    outputs = softmax(outputs)
elif activation == 'relu':
    outputs = np.maximum(0,outputs
elif activation == 'tanh':
    outputs = tanh(outputs)
elif activation == 'leakyrelu':
    outputs = leaky_relu(outputs)
elif activation == 'elu':
    outputs = elu(outputs)
elif activation == "selu":
    outputs = selu(outputs)
return outputs
```

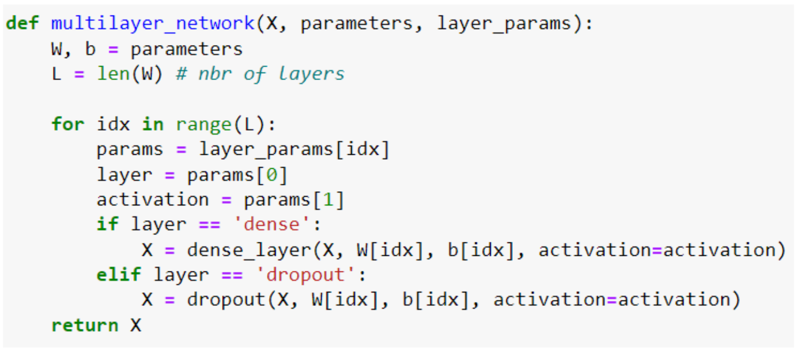
To build a dropout layer, we consider a dense layer and a boolean matrix in which there is a probability p that a value equals to zero. Then we multiply the dense layer results and the random matrix element-wise to put some random results to zero and to keep others. In our case, we considered a dropout rate of 0.5.



```python
def dropout(inputs, weights, bias, activation=None, p=0.5):
    H = dense_layer(inputs, weights, bias, activation=activation)
    U = (np.random.randn(*H.shape) < p) / p
    return H * U
```

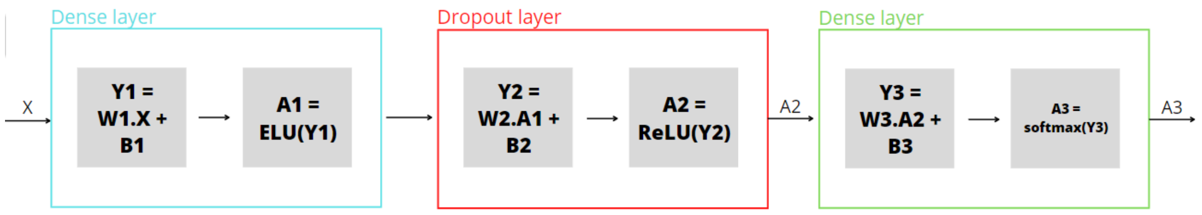
Implemented architecture:

An objective of this project is being able to change the layers of the neural network easily, without having to change everything. Then, we put the layers and activation functions as the parameters of the multilayer_network function.



```python
def multilayer_network(X, parameters, layer_params):
    W, b = parameters
    L = len(W) # nbr of layers

    for idx in range(L):
        params = layer_params[idx]
        layer = params[0]
        activation = params[1]
        if layer == 'dense':
            X = dense_layer(X, W[idx], b[idx], activation=activation)
        elif layer == 'dropout':
            X = dropout(X, W[idx], b[idx], activation=activation)
    return X
```

For this architecture, we implemented a three-layer architecture since it worked well with two. We kept the softmax activation function for the output layer since it is a classification problem. We could also implement an absolute hyperbolic tangent activation function for the output since it is similar to the sigmoid functions. We did not want to put too many layers because it would have resulted in big computation times. We tried to implement the ELU function to see the effect that it had on a neural model and we injected a dropout layer to prevent overfitting.



The corresponding code is:

```
n_input  = 10
n_hidden = [9,5]
n_output = 3
params = initialize_parameters(n_input, n_hidden, n_output)

X = np.random.rand(10, n_input)
layers = [['dense', 'elu'], ['dropout', 'relu'], ['dense', 'softmax']]
print(multilayer_network(X, params,layers))
```

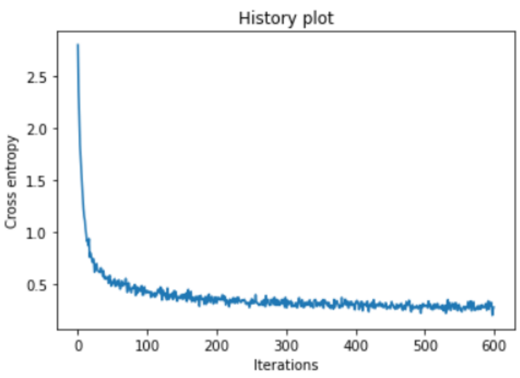
<u>Parameters and training:</u>

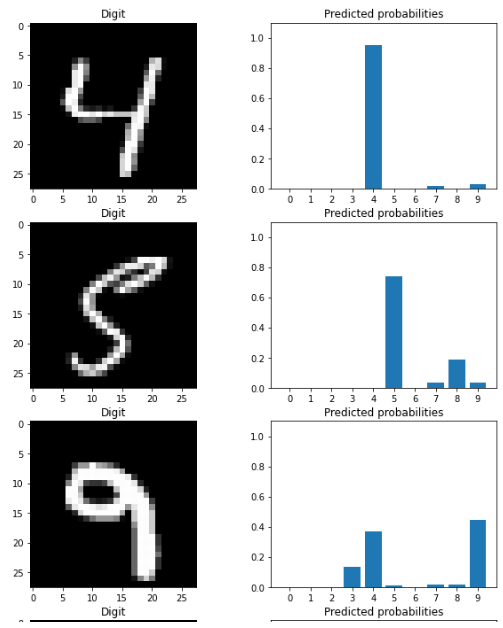
As for the other functions, we must also update the init_parameters function. Indeed, it was configured to take into account only two layers. Here we would like to consider an unlimited number of layers. We must take as arguments the input and output sizes of the data but also a list of size for the hidden layers. From those dimensions, we can generate the different weight and bias matrices.

```
def initialize_parameters(n_input, n_hiddens, n_output):
    """
    Arguments:
    n_input  -- Size of network input
    n_hidden -- Size of hidden layer (1st layer)
    n_output -- Size of network output (2nd layer)

    Returns:
    parameters -- array of arrays W1, b1, W2, b2
    """
    L = len(n_hiddens)
    n1 = n_input
    W = []
    b = []

    idx = 0
    while(idx < L):
        n2 = n_hiddens[idx]
        W.append(np.random.randn(n1, n2) * np.sqrt(2./n1))
        b.append(np.zeros((1,n2)))
        idx+=1
        n1 = n2
    n2 = n_output

    W.append(np.random.randn(n1, n2) * np.sqrt(2./n1))
    b.append(np.zeros((1,n2)))

    return W, b
```

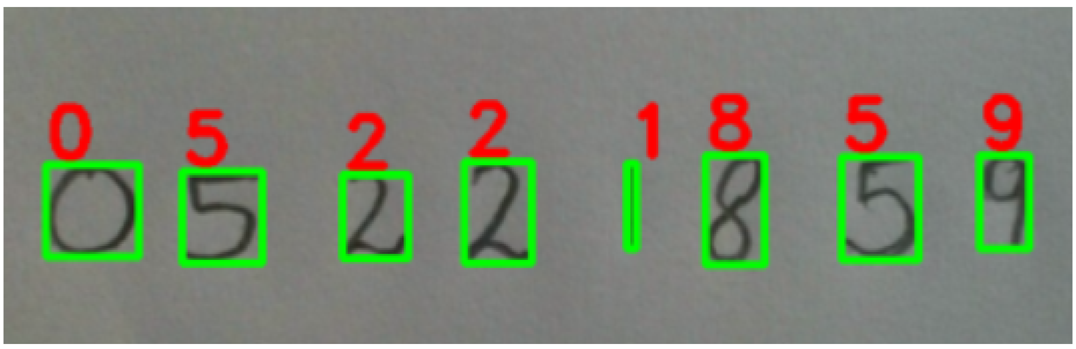
Lastly, since we changed the parameters, we must also make some small changes on the computation of the stochastic training. We must change the two-layer model into a multi-layer one. It is important to pass the layers as parameters of all the training functions so that the multi-layer model wanted is well considered.

<u>Results:</u>





By looking at these graphs, we can see that the convergence is fine and near from 0.5. The probabilities are not as distinct and accurate as for the two-layer model but it still looks great.



We observe that all the characters have been well predicted. However, when testing for other pictures, we found out that there were some mistakes. Indeed, the computed accuracy for this model is 91.3%. We can ask ourselves why this model is less accurate than the two-layer one. An explanation would be that we defined the layers more or less randomly either for the number or neurons or for the type or the activation function. With more time, we would have been able to improve those results by testing a lot of different architectures.

Problems encountered:

We had some problems with the activation functions since the values in parameters were sometimes float and sometimes Autobox types. Indeed, the autograd method returns an Autobox type containing the float value. It took us some time to find out where the problem was. Also, we had some issues with the architecture. For example we did not realize that it was important to put a sigmoid function as the last layer's activation function in classification problems.

Convolutional neural networks:

New layers:

- Convolution layer

In image processing, the goal of a convolution layer is to transform all the pixels from a defined field into a single value. It strongly reduces the dimension of the data. For this architecture, we will consider the following linear operation: $Y = convolution(A, W) + b$. The activation function here is the ReLU function. We could also change it to ELU or SeLU functions since we know that they perform better than the ReLU function.

```python
def convolution_layer(input, weights, bias):
    return relu(np.scipy.convolve(input, weigths) + bias)
```

- Pooling layer

The only use for the pooling layer is to reduce the input dimensions. It does not have to learn any parameters. There exist a bunch of different pooling layers. Here we chose arbitrarly to implement a max-pooling layer. However, we must be careful about the shape of the data. Indeed, the pooling layer will follow a convolutional layer whose output is a 4D-array. We must consider the four dimensions while reshaping the data.

```python
def pooling_layer(input):
    n, d, h, w = data.shape
    data = data.reshape(n, d, h//2, w).max(axis=-3)
    return data.reshape(n, d, h//2, w//2).max(axis=-4)
```

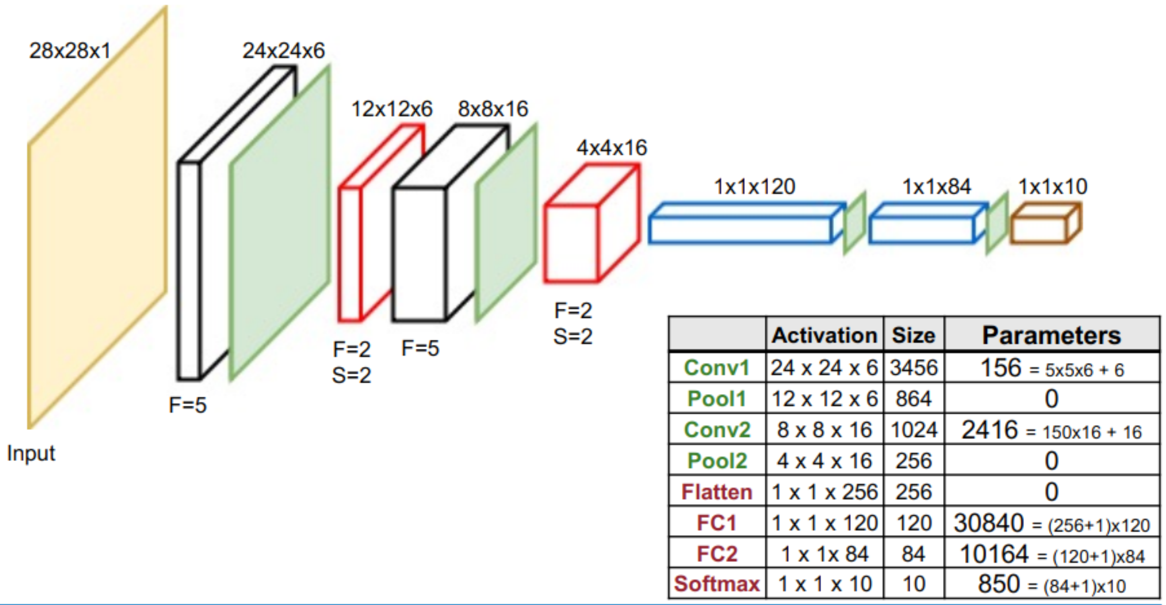
- Flatten layer

This layer has as objective to transform a 4D-array into a 2D-array of size $(n\_images, depth * height * width)$. It always comes before the first dense layer.

```python
def flatten_layer(input):
    n, d, h, w = data.shape
    return data.reshape(n, d*h*w)
```

<u>Convolutional network:</u>

Convolutional networks are often used in image processing. Its first layer is always a convolution layer. In our case, we will follow the architecture of the LeNet5 network. This architecture was proposed by Yann Lecun in 1998, a former student at ESIEE Paris. It is composed of 8 layers:



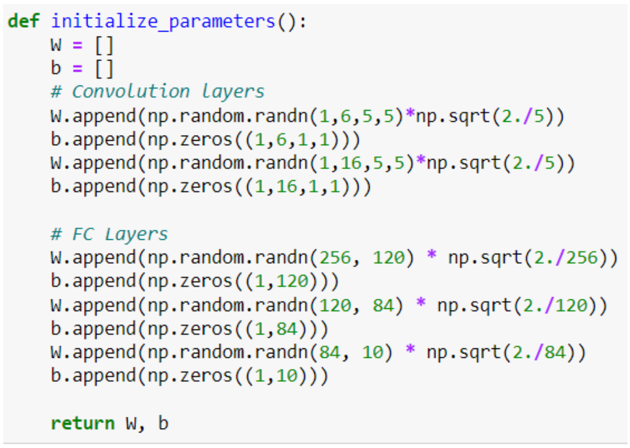
	Activation	Size	Parameters
Conv1	24 x 24 x 6	3456	156 = 5x5x6 + 6
Pool1	12 x 12 x 6	864	0
Conv2	8 x 8 x 16	1024	2416 = 150x16 + 16
Pool2	4 x 4 x 16	256	0
Flatten	1 x 1 x 256	256	0
FC1	1 x 1 x 120	120	30840 = (256+1)x120
FC2	1 x 1x 84	84	10164 = (120+1)x84
Softmax	1 x 1 x 10	10	850 = (84+1)x10

By default, we choose ReLU as activation functions for the two dense layers. The last layer still has a softmax output function since we are still working on a classification problem.

```python
def convolutional_network(X, parameters):
    W, b = parameters
    A0 = pooling_layer(convolution_layer(X, W[0], b[0]))
    A1 = pooling_layer(convolution_layer(A0, W[1], b[1]))
    F = flatten_layer(A1)
    A2 = dense_layer(F, W[2], b[2], activation='relu')
    A3 = dense_layer(A2, W[3], [3], activation='relu')
    return dense_layer(A3, W[4], b[4], activation='softmax')
```

Parameters initialization:

For this function, it is important to distinguish between the convolution layers and the dense layers. Indeed, the weight and bias matrices are not the same depending on the type of the layer. The dimensions for a convolution layer's weight and bias matrices are $(channels,\ filters,\ size,\ size)$ and $(1,\ filters,\ 1,\ 1)$. For a dense layer, it is the same dimensions as seen previously. Then, it was easy to create the matrices from LeNet5 Architecture.



```python
def initialize_parameters():
    W = []
    b = []
    # Convolution layers
    W.append(np.random.randn(1,6,5,5)*np.sqrt(2./5))
    b.append(np.zeros((1,6,1,1)))
    W.append(np.random.randn(1,16,5,5)*np.sqrt(2./5))
    b.append(np.zeros((1,16,1,1)))

    # FC Layers
    W.append(np.random.randn(256, 120) * np.sqrt(2./256))
    b.append(np.zeros((1,120)))
    W.append(np.random.randn(120, 84) * np.sqrt(2./120))
    b.append(np.zeros((1,84)))
    W.append(np.random.randn(84, 10) * np.sqrt(2./84))
    b.append(np.zeros((1,10)))

    return W, b
```

Results:

Unfortunately, we could not obtain results because we had some issues with the sizes of the weight and bias matrices. We tried to load a MNIST dataset with Keras and to reshape and preprocess the data following a tutorial but some issues still remain. With more time maybe we could finish this project completely.

Bibliography:

- Giovanni CHIERCHIA , Deep Learning Lectures
- https://sefiks.com/2017/01/29/hyperbolic-tangent-as-neural-network-activation-function/#:~:text=Hyperbolic%20Tangent%20Function%3A%20tanh
- http://www.gabormelli.com/RKB/Leaky_Rectified_Linear_Activation_(LReLU)_Function
- https://iq.opengenus.org/scaled-exponential-linear-unit/
- https://deeplearninguniversity.com/elu-as-an-activation-function-in-neural-networks/
- https://www.databricks.com/glossary/convolutional-layer
- https://en.wikipedia.org/wiki/LeNet
- https://www.datacamp.com/tutorial/convolutional-neural-networks-python