TP 4-6 - Neural Networks in Pure NumPy

The objective of this subject is to make you practice and familiarize with the NumPy library : creating and manipulating multi-dimensional arrays, vectorized computation instead of for-loops, aggregations and linear algebra.

Three sessions are planned for this subject, and you will need to work at home to complete it. You will send the source code of your solution (properly presented and commented), as well as a short written report with answers to the questions and your observations.

1 Objectives

The goal of this TP is to implement a *multi-layer feed-forward neural network* (FFNN), and to apply it to the classification of handwritten digits. The FFNN takes as input 28x28 grey-level images, and output a probability distribution over the ten digits. The predicted digit is then simply the digit with highest probability.

As this course is not about neural networks but on NumPy multi-dimensional arrays and vectorized computation, all FFNN-related computations will be made explicit with mathematical formulas, and your objective is to implement them entirely with NumPy features, without ever using `for`-loops. The `for`-loops will only be used for the main training loop.

2 The MNIST Dataset

The MNIST database (Modified National Institute of Standards and Technology) is a large database of handwritten digits, which has been very often used for the evaluation of machine learning systems. Each handwritten digit is represented by a 28x28 greyscale bitmap, and is labelled by one of the ten digits. Figure 1 shows 9 examples of such handwritten digits.

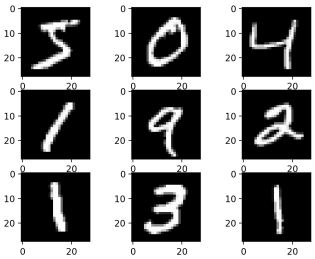

Figure 1 – Excerpt of the MNIST database

The training set contains 60,000 instances, and the test set contains 10,000 instances. The former is used to train the FFNN, while the later (independant of the former) is used to evaluate the trained FFNN. The MNIST database is available on Teams as 4 idx files :

— `train-images.idx3-ubyte` : train inputs
— `train-labels.idx3-ubyte` : train labels
— `t10k-images.idx3-ubyte` : test inputs
— `t10k-labels.idx3-ubyte` : test labels

The best performance, as reported in Wikipedia [1] is an error rate of 0.17% using 20 convolutional neural networks plus a number of improvements. Using a FFNN and a basic learning strategy, we will rather get an error rate around 2.7%, which is quite honorable for such a simple approach.

3 Work Plan

We provide in the following step-by-step instructions to achieve our goal, from reading the MNIST files to the training and evaluation of the FFNN.

3.1 Reading MNIST files

We provide a function `read_idx(`*filepath*`)` that takes as input the filepath to an idx file, and returns a ndarray.

Task 1 *Read the four MNIST files with the given function, and put the results in variables* `x_train`*,* `y_train`*,* `x_test`*, and* `y_test`*. A common convention in machine learning is to use the letter x for inputs, and y for expected outputs (labels).*

idx files can contain ndarrays of various shapes and data types. Run your program, and use the interpreter to answer the following questions.

Question 1 *What are the shapes of the four computed ndarrays ? What is the meaning of the axes of each ndarray ?*

Before working with data, it is good practice to first have a look at them.

Task 2 *Inspect the contents of the data by displaying small slices of the read ndarrays. As inputs contain bitmap images (as 2D ndarrays), they can be visualized with function* `plt.imshow(`2*darray*`)`*.*

3.2 Data preprocessing

The data has to be preprocessed in order to match the expected input and output of the FFNN :

— A FFNN expects vectors (1D arrays) as **inputs**, not 2D arrays. Moreover, the individual values need to be normalized between 0 and 1 in order to avoid problems with large values.

1. `https://en.wikipedia.org/wiki/MNIST_database`

— A FFNN expects vectors as **outputs**, not categorical values. Therefore digit labels (between 0 and 9) need to be replaced by vectors of size 10. For example, label 7 need to be replaced by vector $[0, 0, 0, 0, 0, 0, 0, 1, 0, 0]$ (this is called a one-hot vector), which says that all the probability mass is on digit 7.

For recall, you may only use vectorized computations, and should avoid for-loops.

Task 3 *Define a function for preprocessing inputs (flattening to 1D + normalization), and apply it to train and test inputs to define the FFNN inputs.*
Hint : the transformation must apply to the set of images (e.g., `x_train`), not to a single image.

Task 4 *Define a function for preprocessing outputs (categorical to one-hot vectors), and apply it to define the expected FFNN outputs.*
Hint : use `np.zeros` and fancy indexing on the two axes.

Do not forget to check the results of your preprocessing before proceeding.

3.3 Defining the FFNN components

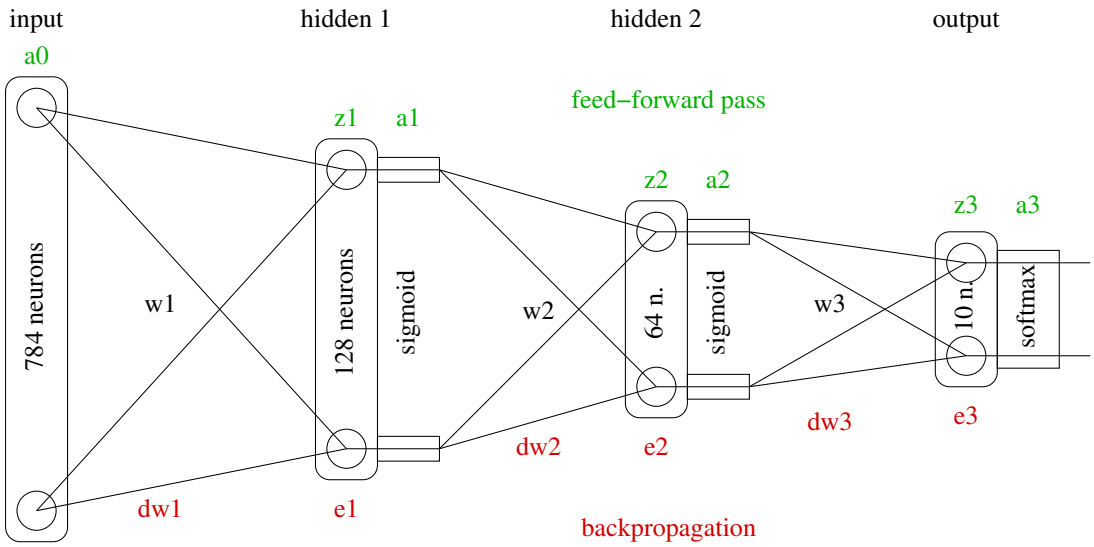

FIGURE 2 – Architecture of our FFNN (black) along with computed values for the feed-forward pass (green), and backpropagation (red).

Figure 2 shows the architecture of our FFNN (Feed-Forward Neural Network). It has one input layer with 784 neurons (one for each pixel of the 28x28 images), one output layer with 10 neurons (one for each digit), and 2 hidden layers in between with respectively 128 and 64 neurons. Each layer is *fully-connected* to the next layer, which means that every neuron of the layer is connected to every neuron of the next layer. Each connection has its own *weight*, a number measuring the strength of the connection.

Task 5 *Define a function* `layer_weights(m,n)` *that returns the set of weights between two layers as a matrix with shape* (m, n)*, where* m *is the number of neurons in the first layer, and* n *is the number of neurons in the second layer.*

The weights must be initialized according to the standard normal distribution, and scaled by the factor $\frac{1}{\sqrt{n}}$*.*

Task 6 *Introduce three new variables* `w1`*,* `w2`*, and* `w3` *for the three matrices of weights, and initialize them using the function defined in the previous task.*

We need two additional components for our FFNN : functions *sigmoid* and *softmax*. The *sigmoid* function, often denoted by S, is defined for any scalar x by

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

The *softmax* function, often denoted by σ, normalizes any vector into a probability distribution. It is defined for any vector $\mathbf{x}$ by

$$\sigma(\mathbf{x})_i = \frac{e^{(x_i - X)}}{\sum_j e^{(x_j - X)}} \quad \text{where } X = max_j\ x_j.$$

We will also need for backpropagation the derivative of the sigmoid function.

$$S'(x) = \frac{e^{-x}}{(e^{-x} + 1)^2}.$$

Task 7 *Define the three above functions in Python. Define functions* S *and* S' *with ufuncs so that they can be applied element-wise to ndarrays.*

3.4 Feed-foward pass : Using the FFNN

The feed-forward pass propagates an input (here an image) from the input layer to the output layer through the two hidden layers.

Each layer i has an activation array a_i with shape $(n_i,)$ that defines an *activation value* for each of the n_i neurons in the layer. Except in the input layer, activation values a_i are the result of applying the sigmoid or the softmax function to the same-shape array z_i. Each value in z_i is a weighted sum involving activation values a_{i-1} of the previous layer and the weight matrix w_i. The equations relating all those values are the following, where $\cdot$ denotes the dot product, and where a_o is initialized as an input taken from training data (for training), or from test data (for evaluation) :

$$\begin{aligned} z_1 &= a_o \cdot w_1 \qquad & a_1 &= S(z_1) \\ z_2 &= a_1 \cdot w_2 \qquad & a_2 &= S(z_2) \\ z_3 &= a_2 \cdot w_3 \qquad & a_3 &= \sigma(z_3) \end{aligned}$$

The output of the network is the activation array a_3.

Question 2 *Analyze the equations in terms of array shapes in order to verify that they are well-formed.*

Task 8 *Define a function* `forward_pass` *that takes an input as argument, and returns the output as a result.*

Task 9 *Extract an input from the training data, and test your* `forward_pass` *function to it.*

3.5 Backpropagation : Improving the FFNN

Training a FFNN amounts to apply small changes δw_i to the weights w_i in order to reduce errors observed between the output a_3 of the FFNN and the expected output y. Those changes are computed by propagating error values e_i backward in the network, starting with the output layer, and going through the two hidden layers (see red annotations in Figure 2).

We apply backpropagation to the same input and output than the feed-forward pass. As error values are compared to activation values, each array e_i has the same shape as a_i. As weight changes are applied to weight values, each array δw_i has the same shape as w_i. Here are the equations defining error values and weight changes as a function of the values computed during the feed-forward pass, and the expected output y. Symbol $\odot$ denotes point-wise multiplication, and a^T is the transpose of matrix a.

$$\begin{array}{ll} e_3 = a_3 - y & \delta w_3 = a_2^T \cdot e_3 \\ e_2 = (e_3 \cdot w_3^T) \odot S'(z_2) & \delta w_2 = a_1^T \cdot e_2 \\ e_1 = (e_2 \cdot w_2^T) \odot S'(z_1) & \delta w_1 = a_0^T \cdot e_1 \end{array}$$

Question 3 *Analyze the equations in terms of array shapes in order to verify that they are well-formed.*

Task 10 *Define a function* `backpropagation` *that takes the values a_i, z_i computed by the feed-forward pass, and the corresponding output y, and returns the three matrices of weight changes.*
Hint : you will need to modify function `forward_pass` so that it returns the intermediate values, not only the outputs.

To complete the handling of a batch, weights need to be updated according to the equation $w_i = w_i - \lambda \cdot \delta w_i$ for $i \in \{1, 2, 3\}$, and where λ is a learning rate (e.g., $\lambda = 0.001$).

Task 11 *Define a function that updates all weight values based on the learning rate, and the weight changes computed during the backpropagation pass.*

3.6 Evaluating the FFNN on test data

In order to evaluate our FFNN, we need to measure its error rate on the test data. The FFNN makes an error on a test instance if the predicted digit differs from the expected digit.

Task 12 *Define a function* `compute_error(x,y)` *returning the error rate on* (x, y) *instances, where* x *is the set of inputs, and* y *is the corresponding set of expected outputs.*
Hint 1 : function `np.argmax` enables to replace an axis by the index of the maximal value.
Hint 2 : vectorized aggregations (like `np.sum`) can be applied to ndarrays of Booleans, which can be seen as ndarrays containing 0s and 1s.

Task 13 *Apply your evaluation function on the test data, using the randomly initialized weights. Is the result conform to your expectation ?*

3.7 Training the FFNN

Now that we have all the necessary pieces, it only remains to put them together to train and evaluate our FFNN. The training is organized as two nested loops. The inner loop iterates over all inputs. For each input, it performs a feed-forward pass to compute an output, then it performs backpropagation to compute weight changes, and finally it applies those changes to the weights. The outer loop simply repeats the inner loop process, called an *epoch*, a fixed number of times. At the end of each epoch, the error rate is computed, and displayed on the console in order to monitor the progress of the training. The message can also display the epoch number, and the runtime.

Task 14 *Define a function* `train` *that implements the process described above, given 2 hyper-parameters :*
- *the number of epochs (default : 30) ;*
- *the learning rate (default : 0.001).*

Task 15 *Call your* `train` *function, and observe how the error rate decreases with each epoch, demonstrating the capability of the FFNN to learn. If each epoch takes too much time, use only a subset of the training set.*

3.8 Using Batches for Efficient Training

As you have probably observed in the previous task, training is very slow, and the error rate remains high. The reason is that the inner loop of the training function has to iterate over many examples (up to 60,000). The solution is to use vectorized computation instead of Python iteration. Concretely, this means that a *batch* of inputs is pushed at once through the FFNN. This requires adding one axis to all arrays : inputs and outputs, activations and errors.

Task 16 *Copy your code in a new file, and modify your functions* `softmax`, `forward_pass`, `backpropagation`, *and* `compute_error` *so that they work with this additional axis running through the batch of training examples.*

Task 17 *Modify you* `train` *function to replace the Python iteration by the new vectorized computation, using the whole training set as a batch. Is training more efficient ? Is the error rate better ? Why ?*

Instead of pushing through the FFNN either a single input or all inputs, we want to choose the size of batches. The training set is therefore cut in as many batches as necessary, and each batch is pushed through the FFNN in turn with a Python loop (much smaller than in the first version).

Task 18 *Introduce a new parameter defining the batch size, and modify the* `train` *function to cut the training sets in batches, and iterate over them. Experiment with different batch sizes. Do you retain efficiency ? Is the error rate improved ?*