Neural Networks for Beginners

A fast implementation in MATLAB, Torch, TensorFlow

F. Giannini[1], V. Laveglia[1,2], A. Rossi[1,3]*, D. Zanca[1,2], A. Zugarini[1]

[1]DIISM, University of Siena, Siena, Italy
[2]DINFO, University of Florence, Florence, Italy
[3]Fondazione Bruno Kessler, Trento, Italy

rossi111@unisi.it
{giannini7, andrea.zugarini}@student.unisi.it
{vincenzo.laveglia, dario.zanca}@unifi.it

March 17, 2017

What is this report about?

This report provides an introduction to some Machine Learning tools within the most common development environments. It mainly focuses on practical problems, skipping any theoretical introduction. It is oriented to both students trying to approach Machine Learning and experts looking for new frameworks.

The dissertation is about Artificial Neural Networks (ANNs [1, 2]), since currently is the most trend topic, achieving state of the art performance in many Artificial Intelligence tasks. After a first individual introduction to each framework, the setting up of general practical problems is carried out simultaneously, in order to make the comparison easier.

Since the treated argument is widely studied and in continuos and fast growing, we pair this document with an on-line documentation available at the Lab GitHub repository [3] which is more dynamic and we hope to be kept updated and possibly enlarged.

*Corresponding Author

Contents

1 MATLAB: a unified friendly environment

1.1 Introduction

MATLAB® [4] is a very powerful instrument allowing an easy and fast handling of almost every kind of numerical operation, algorithm, programming and testing. The intuitive and friendly interactive interface makes it easy to manipulate, visualize and analyze data. The software provides a lot of mathematical built-in functions for every kind of task and an extensive and easily accessible documentation. It is mainly designed to handle matrices and, hence, almost all the functions and operations are vectorized, i.e. they can manage scalars, as well as vectors, matrices and (often) tensors. For these reasons, it is more efficient to avoid loops cycles (when possible) and to set up operations exploiting matrices multiplication.

In this document we just show some simple Machine Learning related instruments in order to start playing with ANNs. We assume a basic-level knowledge and address to official documentation for further informations. For instance, you can find informations on how to obtain the software from the official web site[1]. Indeed, the license is not for free and even if most universities provide a classroom license for students use, maybe could not be possible to access to all the current packages. In particular the Statistic and Machine Learning Toolbox™ and the Neural Network Toolbox™ provide a lot of built-in functions and models to implement different ANNs architectures suitable to face every kind of task. The access to both the tools is fundamental in the prosecution, even if we refer to some simple independent examples. The most easy to-go is the `nnstart` function, which activates a simple GUI guiding the user trough the definition of a simple 2-layer architecture. It allows either to load available data samples or to work with customize data (i.e. two matrices of input data and correspondent target), train the network and analyze the results (Error trend, Confusion Matrix, ROC, etc.). However, more functions are available for specific tasks. For instance, the function `patternnet` is specifically designed for pattern recognition problems, `newfit` is suitable for regression, whereas `feedforwardnet` is the most flexible one and allows to build very customized and complicated networks. All the versions are implemented in a similar way and the main options and methods apply to everyone. In the next section we show how to manage customizable architectures starting to face very basic problems. Detailed informations can be find in a dedicated section of the official site[2].

CUDA® computing

GPU computing in MATLABrequires the Parallel Computing Toolbox™ and the CUDA® installation on the machine. Detailed informations on how to use, check and set GPUs devices can be found in GPU computing official web page[3], where issues on Distributed Computing CPUs/GPUs are introduced too. However, basic operations with graphical cards should in general be quite simple. Data can be moved to the GPU hardware by the function `gpuArray`, then back to the CPU by the function `gather`. When dealing with ANNs, a dedicated function `nndata2gpu` is provided, organizing tensors (representing a dataset) in a efficient configuration on the GPU, in order to speed up the computation. An alternative way is to carry out just the training process in the GPU by the correspondent option of the function `train` (which will be describe in details later). This can

[1] `https://ch.mathworks.com/products/matlab.html?s_tid=hp_products_matlab`

[2] `http://ch.mathworks.com/help/nnet/getting-started-with-neural-network-toolbox.html`

[3] `https://ch.mathworks.com/help/nnet/ug/neural-networks-with-parallel-and-gpu-computing.html`

be done directly by passing additional arguments, in the *Name,Values* pair notation, the option `'useGPU'` and the value `'yes'`:

```
nn = train(nn, ... , 'useGPU','yes')
```

1.2 Setting up the XOR experiment

The XOR is a well-known classification problem, very simple and effective in order to understand the basic properties of many Machine Learning algorithms. Even if writing down an efficient and flexible architecture requires some language expertise, a very elementary implementation can be found in the MATLABsection of the GitHub repository[4] of this document. It is not suitable to face real tasks, since no customizations (except for the number of hidden units) are allowed, but can be useful just to give some general tips to design a personal module. The code we present is basic and can be easily improved, but we try to keep it simple just to understand fundamental steps. As we stressed above, we avoid loops exploiting the MATLABefficiency with matrix operations, both in forward and backward steps. This is a key point and it can substantially affects the running time for large data.

Initialization

Here below, we will see how to define and train more efficient architectures exploiting some built-in functions from the Neural Network Toolbox™ . Since we face the XOR classification problem, we sort out our experiments by using the function `patternnet`. To start, we have to declare an object of kind `network` by the selected function, which contains variables and methods to carry out the optimization process. The function expects two optional arguments, representing the number of hidden units (and then of the hidden layers) and the back-propagation algorithm to be exploited during the training phase. The number of hidden units has to be provided as a single integer number, expressing the size of the hidden layer, or as an integer row vector, whose elements indicate the size of the correspondent hidden layers. The command:

```
nn = patternnet(3)
```

creates on object named `nn` of kind `network`, representing a 2-layer ANN with 3 units in the single hidden layer. The object has several options, which can be reached by the dot notation *object.property* or explore by clicking on the interactively visualization of the object in the MATLABCommand Window, which allows to see all the available options for each property too. The second optional parameter selects the training algorithm by a string saved in the `trainFcn` property, which in the default case takes the value '*trainscg*' (Scaled Conjugate Gradient Descent methods). The network object is still not fully defined, since some variables will be adapted to fit the data dimension at the calling of the function `train`. However, the function `configure`, taking as input the object and the data of the problem to be faced, allows to complete the network and set up the options before the optimization starts.

[4] Available at `https://github.com/AILabUSiena/NeuralNetworksForBeginners/tree/master/matlab/2layer`.

Dataset

Data for ANNs training (as well as for others available Machine Learning methods) must be provided in matrix form, storing each sample column-wise. For example data to define the XOR problem can be simply defined via an input matrix X and a target matrix Y as:

```
X = [0, 0, 1, 1; 0, 1, 0, 1]
Y = [0, 1, 1, 0]
```

MATLABexpects targets to be provided in 0/1 form (other values will be rounded). For 2-class problem targets can be provided as a row vector of the same length of the number of samples. For multi-class problem (and as an alternative for 2-class problem too) targets can be provided in the *one-hot encoding* form, i.e. as a matrix with as many columns as the number of samples, each one composed by all 0 with only a 1 in the position indicating the class.

Configuration

Once we have defined data, the network can be fully defined and designed by the command:

```
nn = configure(nn,X,Y)
```

For each layer, an object of kind `nnetLayer` is created and stored in a *cell* array under the field `layers` of the network object. The number of connections (the weights of the network) for each units corresponds to the layer input dimension. The options of each layer can be reached by the dot notation *object*. `layer{`$numberOfLayer$`}`.*property*. The field `initFcn` contains the weights initialization methods. The activation function is stored in the `transferFcn` property. In the hidden layers the default values is the '*tansig*' (Hyperbolic Tangent Sigmoid), whereas the output layers has the '*logsig*' (Logistic Sigmoid) or the '$softmax$' for 1-dimensional and multi-dimensional target respectively. The '*crossentropy*' penalty function is set by default in the field `performFcn`. At this point, the global architecture of the network can be visualized by the command:

```
view(nn)
```

Training

The function `train` itself makes available many options (as for instance `useParallel` and `useGPU` for heavy computations) directly accessible from its interactive help window. However, it can take as input just the network object, the input and the target matrices. The optimization starts by dividing data in Training, Validation and Test sets. The splitting ratio can be changed by the options `divideParam`. In the default setting, data are randomly divided, but if you want for example to decide which data are used for test, you can change the way the data are distributed by the option `divideFcn`[5]. In this case, because of the small size of the dataset, we drop validation and test by setting:

[5]Click on `divideFcn` property from the MATLABCommand Window visualization of your object to see the available methods.

```
nn.divideFcn = ''
```

In the following code, we set the training function to the classic gradient descent method '*traingd*', we deactivate the training interactive GUI by `nn.trainParam.showWindow` (boolean) and activate the printing of the training state in the Command Window by `nn.trainParam.showCommandLine` (boolean). Also the *learning rate* is part of the `trainParam` options under the fields `lr`.

```
nn.trainFcn = 'traingd'
nn.trainParam.showWindow = 0
nn.trainParam.showCommandLine = 1
nn.trainParam.lr = 0.01
```

Training starts by the calling:

```
[nn, tr] = train(nn, X, Y)
```

this generates a printing, ending in this case with:

```
Epoch 875/1000, Time 1.2259, Performance 0.15525/0, Gradient 0.12428/1e-05, Validation Checks 0/6
Epoch 900/1000, Time 1.2512, Performance 0.15149/0, Gradient 0.12094/1e-05, Validation Checks 0/6
Epoch 925/1000, Time 1.2752, Performance 0.14792/0, Gradient 0.11778/1e-05, Validation Checks 0/6
Epoch 950/1000, Time 1.298, Performance 0.14454/0, Gradient 0.11478/1e-05, Validation Checks 0/6
Epoch 975/1000, Time 1.3326, Performance 0.14133/0, Gradient 0.11193/1e-05, Validation Checks 0/6
Epoch 1000/1000, Time 1.3603, Performance 0.13827/0, Gradient 0.10922/1e-05, Validation Checks 0/6
Training with TRAINGD completed: Maximum epoch reached.
```

This indicates that the training stops after the max number of epoch is reached (which can be set by options *object*. `trainParam.epochs`). Each column shows the state of one of the stopping criterions used, which we will analyze in details in the next section. The output variable *tr* stores the training options. The fields `perf`, `vperf` and `tperf` contain the performance of the network evaluated at each epoch on the Training, Validation and Test sets respectively (the last two are `NaN` in this case), which can be used for example to plot performances. If we pass data organized in a single matrix, the function will exploit the full batch learning method accumulating gradients overall the training set. To set a mini-batch mode, data have to be manually split in sub-matrix with the same number of column and organized in a cell array. However, let us consider for a moment a general data set composed by N samples in the features space $\mathbb{R}^D$ with a target of dimension C, so that $X \in \mathbb{R}^{D\times N}$ and $Y \in \mathbb{R}^{C\times N}$. All the mini-batches have to be of the same size b, so that it is in general convenient to choose the batch size to be a factor of N. In this case, we can generate data for the training function organizing the input and target in the correspondent `cell-array` by:

```
N = size(X,2); % number of samples
n_batch = N/batchsize; % number of batches

input{n_batch} = []; % input cell-array initialization
target{n_batch} = []; % target cell-array initialization

p = randperm(N); % generating a random permutated index for data shuffling
X = X(:,p); % samples permutation
Y = Y(:,p); % target permutaion
```

```
for i=1:n_batch
  input{i} = X(:,(1:batchsize)+(i-1)*batchsize);
  target{i} = Y(:,(1:batchsize)+(i-1)*batchsize);
end
```

However, in order to perform a pure *Stochastic Gradient Descent* optimization, in which the ANNs parameters are updated for each sample, the training function `'trains'` have to be employed skipping to split data previously. A remark has to be done since this particular function does not support the GPU computing.

The network (trained or not) can be easily evaluated on data by passing the input data as argument to a function named as the network object. Performance of the prediction with respect to the targets can be evaluated by the function `perform` according to the correspondent loss function option *object*. `performFcn`:

```
  f = nn(X)
  perform(nn, Y, f)
```

1.3 Stopping criterions and Regularization

Early stopping is a well known procedure in Machine Learning to avoid overfitting and improve generalization. Routines from Neural Network Toolbox™ use different kind of stopping criterions regulated by the network object options in the field `trainParam`. Arbitrarily methods are based on the number of epochs (`epochs`) and the training time (`time`, default Inf). A criterion based on the training set check the loss (*object*. `trainParam.goal`, default = 0) or the parameters gradients (*object*. `trainParam.min_grad`, default = 10^{-6}) to reach a minimum threshold. A general *early stopping* method is implemented by checking the error on the validation set and interrupting training when validation error does not improve for a number of consecutive epochs given by `max_fail` (default = 6).

Further regularization methods can be configured by the property `performParam` of the network object. The field `regularization` contains the weight (real number in $[0, 1]$) balancing the contribution of a term trying to minimizing the norm of the network weights versus the satisfaction of the penalty function. However, the network is designed to mainly rely on the validation checks, indeed regularization applies only to few kind of penalties and the default weight is 0.

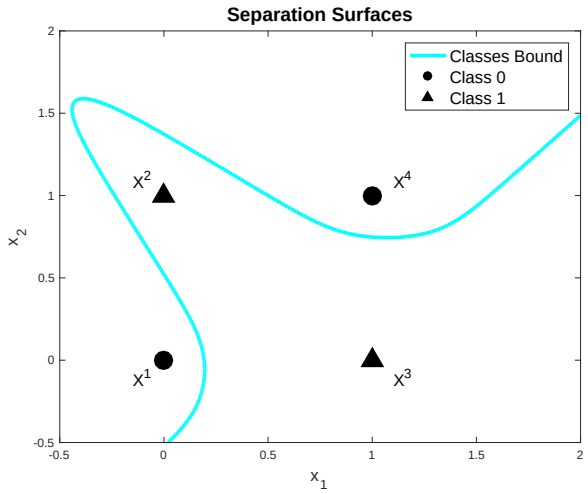
1.4 Drawing separation surfaces

When dealing with low dimensional data (as in the XOR case), can be useful to visualize the prediction of the network directly in the input space. For this kind of task, MATLABmakes available a lot of built-in functions with many options for interactive data visualization. In this section, we will show the main functions useful to realize customized separation surfaces learned by an ANN with respect to some specific experiments. We briefly add some comments for each instruction, referring to the suite help for specific knowledge of each single function. The network predictions will be evaluated on a grid of the input space, generated by the MATLABfunction `meshgrid`, since the main functions used for the plotting (`contour` or, if you want a color surface `pcolor`) require

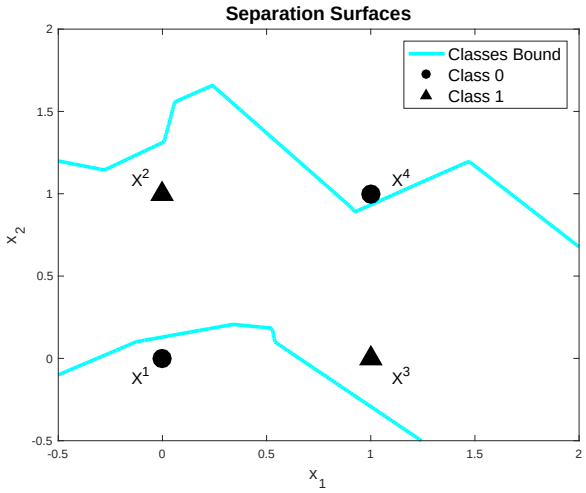
as input three matrices of the same dimensions expressing, in each correspondent element, the coordinates of a 3-D point (which in our case will be first input dimension, second input dimension and prediction). Once we trained the network described until now, the boundary for the 2-classes separation showed in Figure 1a is generated by the code in Listing 1, whereas in Figure 1b we report the same evaluation after the training of a 4-layers network using 5, 3 and 2 units in the first, second and third hidden layers respectively, each one using the *ReLU* as activation (`'poslin'` in MATLAB). This new network can be defined by:

```
n = patternnet([5,3,2]); % 3+output layers network initialization
nn = configure(n,X,Y); % network configuration
nn.trainFcn = 'traingd'; % setting optimization function
nn.divideParam.trainRatio = 1; % setting data splitting ratios (illustrative)
nn.divideParam.valRatio = 0;
nn.divideParam.testRatio = 0;
nn.trainParam.showCommandLine = 1;
nn.trainParam.lr = 0.01;
nn.layers{1}.transferFcn = 'poslin'; % setting the activation layer-wise
nn.layers{2}.transferFcn = 'poslin';
nn.layers{3}.transferFcn = 'poslin';
```

Listing 1: Drawing separation surfaces



(a) 2-layers, 3 Hidden Units



(b) 4-layers, 5, 3 and 2 Hidden Units, ReLU for all the activations

Figure 1: Separation surfaces on the XOR classification task.

```
%%%%  Plotting Separation Surface

% generating input space grid
[xp1,xp2] = meshgrid(-0.5:.01:2,-0.5:.01:2);
% network evaluation on the gridding (reshaped to fit network input dimension)
f = nn([xp1(:)';xp2(:)']);
% reshaping prediction in correspondent matrix form
f = reshape(f,size(xp1,1),[]);
```

```
9
% drawing separation surfaces
11 contour(xp1,xp2,f,[.5,.5],'LineWidth',3,'Color','c');
hold on;
13
% drawing data points
15 scatter(X(1,[1,4]),X(2,[1,4]),200,'o','filled','MarkerEdgeColor','k',...
'MarkerFaceColor','k','LineWidth',2);
17 scatter(X(1,[2,3]),X(2,[2,3]),200,'^','filled','MarkerEdgeColor','k',...
'MarkerFaceColor','k','LineWidth',2);
19
axis([-0.5,2,-0.5,2]); % setting axis bounds
21
% labeling data points
23 c = {'X^1','X^2','X^3','X^4'}; % labels
dx = [-.15, -.15, .1, .1]; % labels horizontal translation wrt points
25 dy = [-.1, .1, -.1, .1]; % labels vertical translation wrt points
text(X(1,:)+dx, X(2,:)+dy, c, 'FontSize',14); % showing labels as text
27
% plot labels
29 xlabel('x_1','FontSize',14)
ylabel('x_2','FontSize',14)
31 title('Separation Surfaces','FontSize',16);

33 h = legend({'Classes Bound','Class 0','Class 1'},'Location','NorthEast');

35 set(h,'FontSize',14);
```

2 Torch and Lua environment

2.1 Introduction

Torch7 is an easy to use and efficient scientific computing framework, essentially oriented to Machine Learning algorithms. The package is written in C which guarantees an high efficiency. However, a completely interaction is possible (and usually convenient) by the *LuaJIT* interface, which provides a fast and intuitively scripting language. Moreover, it contains all the libraries necessary for the integration with the CUDA® environment for GPU computing. At the moment of writing it is one of the most used tool for prototyping ANNs of any kind of topology. Indeed, there are many packages, constantly updated and improved by a large community, allowing to develop almost any kind of architectures in a very simple way.

Informations about the installation can be found at the *getting started* section of the official site[6]. The procedure is straightforward for *UNIX* based operative systems, whereas is not officially supported for Windows, even if an alternative way is provided[7]. If CUDA® is already installed, also the packages `cutorch` and `cunn` will be added automatically, containing all the necessary utilities to deal with Nvidia GPUs.

2.2 Getting started

2.2.1 Lua

Lua, in torch7, acts as an interface for C/CUDA routines. A programmer, in most of the cases, will not have to worry about C functions. Therefore, we explain here only how the Lua language works, because is the only one necessary to deal with Torch. It is a scripting language with a syntax similar to Python and semantic close to Javascript. A variable is considered as *global* by default. The local declaring, which is usually recommended, require the explicit declaration by placing the keyword `local` before the name fo the variable. Lua has been chosen over other scripting languages, such as Python, because is the fastest one, a crucial feature when dealing with large data and complex programs, as common in Machine Learning.

There are seven native types in lua: `nil, boolean, number, string, userdata, function` and `table`, even if most of the Lua power is related to the last one. A `table` behaves either as an hash map (general case) or as an array (which have the 1-based indexing as in MATLABand Python). The table will be considered as an array when contains only numerical keys, starting from the value 1. Any other complex structure such as classes, are built from it (formally defined as a *Metatable*).

A detailed documentation on Lua can be find at the official webpage[8], however, an essential and fast introduction can be found at `http://tylerneylon.com/a/learn-lua/`.

2.2.2 Torch enviroment

Torch extends the capabilities of the Lua `table` implementing the `Tensor` class. Many Matlab-like functions[9] are provided in order to initialize and manipulate tensors in a concise fashion. Most

[6] `http://torch.ch/docs/getting-started.html`
[7] `https://github.com/torch/torch7/wiki/Windows`
[8] Lua 5.1 reference manual is available here: `https://www.lua.org/manual/5.1/`
[9] `http://atamahjoubfar.github.io/Torch_for_Matlab_users.pdf`

commons are reported in Listing 2.

```
local t1 = torch.Tensor() -- no dimension tensor constructor
local t2 = torch.Tensor(4,3) -- 4x3 empty tensor
local t3 = torch.eye(3,5) -- 3x5 1-diagonal matrix
t2:fill(1) -- fill the matrix with the value 1
t1 = torch.mm(t2,t3) -- assign to t1 the result of matrix multiplication between t2
    and t3
t1[1][2] = 5 -- assign 5 to the element in first row and second column
```

Listing 2: Example of torch tensor basic usages

All the provided packages are developed following a strong modularization, which is a crucial feature to keep the code stable and dynamic. Each one provides several already built-in functionalities, and all of them can be easily imported from Lua code. The main one is, of course, `torch`, which is installed at the beginning. Not all the packages are included at first installation, but it is easy to add a new one by the shell command:

```
luarocks install packagename
```

where `luarocks` is the package manager, and `packagename` is the name of the package you want to install.

The `nn` package

All (almost) you need to create (almost) any kind of ANNs is contained in the `nn` package (which is usually automatically installed). Every element inside the package inherits from the abstract Lua class `nn.Module`. The main state variables are `output` and `gradInput`, where the result of forward and backward steps (in back-propagation) will be stored. `forward` and `backward` are methods of such class (which can be accessed by the *object:method()* notation). They invoke `updateOutput` and `updateGradInput` respectively, that here are abstract and the definition must be in the derived classes.

The main advantage of this package is that all the gradients computations in the back-propagation step are automatically realized thanks to these built-in functions. The only requirement is to call the `forward` step before the `backward`.

The weights of the network will be updated by the method `updateGradParameters`, assigning a new value to each parameter of a module (according to the Gradient Descent rule) exploiting the *learning rate* passed an argument of the function.

The bricks you can use to construct a network can be divided as follows:

- **Simple layers:** the common modules to implement a layer. The main is `nn.Linear`, computing a basic linear transformation.
- **Transfer functions:** here you can find many activation functions, such as `nn.Sigmoid` or `nn.Tanh`
- **Criterions:** loss functions for supervised tasks, a for instance is `nn.MSECriterion`
- **Containers:** abstract modules that allow us to build multi-layered networks. `nn.Sequential` connect several layers in a feed-forward manner. `nn.Parallel` and `nn.Concat` are important to build more complex structure, where the input flows in separated architectures. Layers, activation functions and even criterions can be added inside those containers.

For detailed documentation of the `nn` package we refer to the official webpage[10]. Another useful package for whom could be interested on building more complex architectures can be found at the `nngraph` repository[11].

CUDA® computing

Since C++/Cuda programming and integration are not trivial to develop, it is important to have an interface as simple as possible linking such tools. Torch provides a clean solution for that with the two dedicated packages `cutorch` and `cunn` (requiring, of course, a capable GPU and CUDA® installed). All the objects can be transferred into the memory of GPUs by the method `:cuda()` and then back to the CPU by `:double()`. Operations are executed on the hardware of the involved objects and are possible only among variables from the same unit. In Listing 3 we show some examples of correct and wrong statements.

```
local cpuTensor1 = torch.Tensor(3,3):fill(1)
local cpuTensor2 = torch.Tensor(3,3):fill(2)
local cudaTensor1 = torch.Tensor(3,3):fill(3):cuda()
local cudaTensor2 = torch.Tensor(3,3):fill(4):cuda()

cpuTensor1:cmul(cpuTensor2) -- OK
cpuTensor1:cmul(cudaTensor2) -- WRONG
cudaTensor1:cmul(cudaTensor2) -- OK
cudaTensor1:cmul(cpuTensor2) -- WRONG
```

Listing 3: Samples of CUDA operations.

2.3 Setting up the XOR experiment

In order to give a concrete feeling about the presented tools, we show some examples on the classical XOR problem as in the previous section. The code showed here below can be found in the Torch section of the document's GitHub repository[12] and can be useful to play with the parameters and become more familiar with the environment.

Architecture

When writing a a script, the first command is usually the import of all the necessary packages by the keyword `require`. In this case, only the `nn` toolbox is required:

```
require 'nn'
```

We define a standard ANNs with one hidden layer composed by 2 hidden units, the *hyperbolic tangen* (`tanh`) as transfer function and identity as output function. The structure of the network will be stored in a container where all the necessary modules will be added. A standard feed-forward architecture can be defined into a `Sequential` container, which we named *mlp*. The network can be then assembled by adding sequentially all the desired modules by the function `add()`:

[10] `https://github.com/torch/nn`
[11] Detailed documentation at `https://github.com/torch/nngraph`
[12] Available at `https://github.com/AILabUSiena/NeuralNetworksForBeginners/tree/master/torch/xor`.

```
mlp = nn.Sequential() -- container initialization
inputs = 2; outputs = 1; HUs = 2; -- general options
mlp:add(nn.Linear(inputs, HUs)) -- first linear layer
mlp:add(nn.Tanh()) -- activation for the hidden layer
mlp:add(nn.Linear(HUs, outputs)) -- output layer
```

Listing 4: Code to create a 2-layer ANNs

Dataset

The training set will be composed by a tensor of 4 samples (organized again column-wise) paired with a tensor of targets. Usually, *true* and *false* boolean values are respectively associated to 1 and 0. However, just to propose an equivalent but different approach, here we shift both values by -0.5, so they will be in $[-0.5, 0.5]$ as showed in Listing 5. Both input and target are initialized with *false* values (a tensor filled with 0), and then *true* values are placed according to the XOR truth table.

```
local dataset = torch.Tensor(4,2):fill(0)
local target = torch.Tensor(4,1):fill(0)
dataset[2][1] = 1 -- True False
dataset[3][2] = 1 -- False True
dataset[4][1] = 1; dataset[4][2] = 1 -- True True
dataset = dataset:add(-0.5) -- shift true and false by -0.5
target[2][1] = 1; target[3][1] = 1
target = target:add(-0.5)
```

Listing 5: creation of 4 examples and their targets

Training

We set up a full–batch mode learning, i.e. we update the parameters after accumulating the gradients over the whole dataset. We exploit the following function:

`forward(`*input*`)` returns the output of the multi layer perceptron w.r.t the given input; it updates the input/output states variables of each modules, preparing the network for the backward step; its output will be immediately passed to the loss function to compute the error.

`zeroGradParameters()` resets to null values the state of the gradients of the all the parameters.

`backward(`*gradients*`)` actually computes and accumulates (averaging them on the number of samples) the gradients with respect to the weights of the network, given the data in input and the gradient of the loss function.

`updateParameters(`*learningrate*`)` modifies the weights according to the Gradient Descent procedure using the learning rate as input argument.

As loss function we use the Mean Square Error, created by the statement:

```
criterion = nn.MSECriterion()
```

When a criterion is forwarded, it returns the error between the two input arguments. It updates its own modules state variable and gets ready to compute the gradients tensor of the loss in the backward step, which will be back-propagated through the multilayer perceptron. As a `nn` modules, all the possible criterions used the functions `forward()` and `backward()` as the others. The whole training procedure can be set up by:

```
nepochs = 1000; learning_rate = 0.05; -- general options
local loss = torch.Tensor(nepochs):fill(0); -- training loss initialization
for i = 1,nepochs do -- iterating over 1000 epochs
  local input = dataset
  local outputs = mlp:forward(input) -- network forward step
  loss[i] = criterion:forward(outputs, target) -- error evaluation
  mlp:zeroGradParameters() -- zero reset of gradients
  local gradients = criterion:backward(mlp.output, target) -- loss gradients
  mlp:backward(input, gradients) -- network backward step
  mlp:updateParameters(learning_rate) -- update parameters given the learning rate
end
```

Listing 6: training of the network

2.4 Stopping criterions and Regularization

Since the training procedure is manually defined, particular stopping criterion are completely up to the user. The simplest one, based on the reaching of a fixed number of epochs explicitly depends of the upper bound of the `for` cycle. Since other methods are related to the presence of a validation set, we will define an example of early stopping criterion in Listing 14 in Section 4. A simple criterion based on the vanishing of the gradients can be simply set up by exploiting the function `getParameters` defined for the modules of `nn`, which returns all the weights and the gradients of the network in two 1-Dimensional vector:

```
param, grad = mlp:getParameters()
```

A simple check on the minimum value of the absolute values of gradients saved in `grad` can be used to stop the training procedure.

Another regularization method can be accomplished by implementing the weight decay method as shown in Listing 7. The presented code is intended to be an introductory example even to understand the class inheritance mechanisms in Lua and Torch.

```
-- defining class inheritances
local WeightDecay, parent = torch.class('nn.WeightDecayWrapper', 'nn.Sequential')

function WeightDecay:__init() -- constructor
  parent.__init(self)
  self.weightDecay = 0 -- self is a keyword referring to the abstract object
  self.currentOutput = 0
end

function WeightDecay:getWeightDecay(alpha)
  local alpha = alpha or 0
  local  weightDecay = 0
  for i=1,#self.modules do
```

```
    local params,_ = self.modules[i]:parameters()
    if params then
      for j=1,#params do
        weightDecay = weightDecay + torch.dot(params[j], params[j])*alpha/2
      end
    end
  end
  self.weightDecay = weightDecay
  return self.weightDecay
end

function WeightDecay:updateParameters(learningRate,alpha)
  local alpha = alpha or 0
  for i=1,#self.modules do
    local params, gradParams = self.modules[i]:parameters()
    if params then
      for j=1,#params do
        params[j]:add(-learningRate, gradParams[j] + (alpha*params[j]))
      end
    end
  end
end
```

Listing 7: Code to implement the weight decay regularization

To implement the weight decay coherently with the `nn` package, we need to create a novel class, inheriting from `nn.Sequential`, that overloads the method `updateParameters()` of the `nn.Module`. We first have to declare a new class name in Torch, where you can optionally specify the parent class. In our case the new class has been called `nn.WeightDecayWrapper`, while the class it inherits from is the Container `nn.Sequential`. The constructor is defined within the function `WeightDecay:__init()`. In the scope of this function the variable `self` is a table used to refer to all the attributes and methods of the abstract object. The calling of the function `__init()` from the `parent` class automatically add all the original properties. The functions `WeigthDecay:getWeigthDecay()` and `WeigthDecay:updateParameters()` compute respectively the weight decay and its gradient. Both methods loop over all the modules of the container (the symbol # returns the number-indexed element of a table) and, for each one that has parameters, use them in order to compute either the error or the gradients coming from the weight decay contribution. The argument `alpha` represent the regularization parameter of the weight decay and, if not provided, is assumed null. It is also worth to mention the fact that, `WeigthDecay:updateParameters()` overloads the method that implemented in `nn.Module`, updating the parameters according to the standard Gradient Descent rule. At this point, an ANN expecting a possible weight decay regularization can be declared by replacing the `nn.Sequential` container by the proposed `nn.WeightDecayWrapper`.

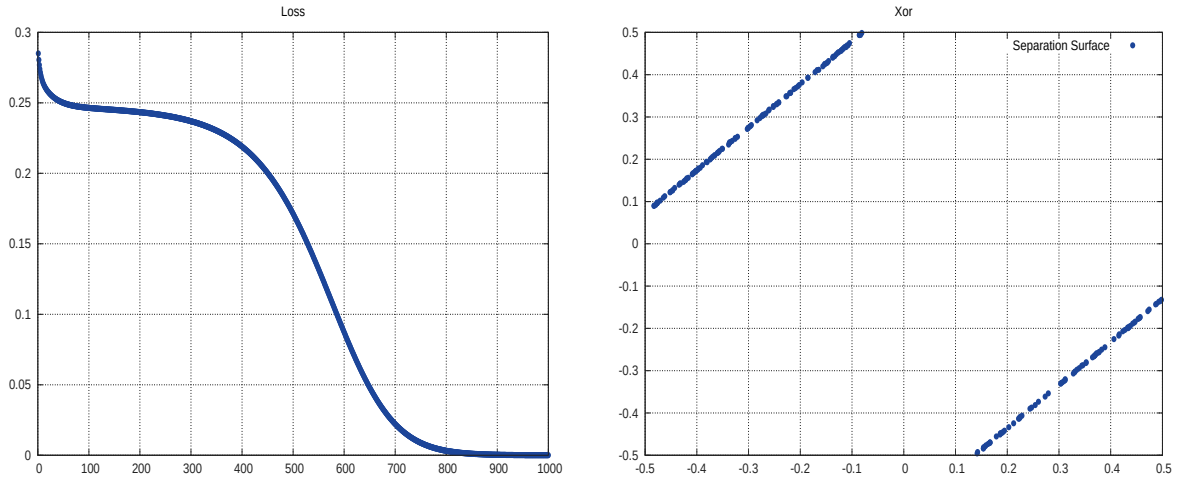
2.5 Drawing Separation Surfaces

In this framework, data visualization is allowed by the package `gnuplot`, which provides some tools to plot points, lines, curves and so on. For example, in the training procedure presented in Listing 6, a vector storing the penalty evaluated at each epoch is produced. To have an idea of the state of the network during training, we can save an image file containing the trend of the error by the code in Listing 8, whom output is shown in Figure 2(a).

```
gnuplot.epsfigure('XORloss.eps') -- create an eps file with the image
gnuplot.plot({'loss function', torch.range(1,nepochs),loss}) -- loss trend plot
gnuplot.title('Loss')
gnuplot.grid(true)
gnuplot.plotflush()
gnuplot.figure()
```

Listing 8: Error evaluated at each training epoch.

Torch does not have dedicated functions to visualize separation surfaces produced on data and, hence, we generate a random grid across the input space, plotting only those points predicted close enough (with respect to a a certain threshold) to the half of possible target (0 in this case). The correspondent result, showed in Figure 2(b), is generated by the code in Listing 9, exploiting the fact that Lua support the logical indexing as in Matlab.



(a) (b)

Figure 2: (a) Trend of loss versus the number of epochs. (b) The estimated separation surface obtained by a 2-Layers ANN composed by 2 Hidden Units, *Hyperbolic Tangent* as activation and linear output.

```
local eps = 2e-3; -- separation threshold
local  n = 1000000 -- gridding size
-- input space grid by n 2D-points uniformly distributed in [-0.5,0.5]
local x = torch.rand(n,2):add(-0.5)
local mlpOutput = mlp:forward(x) -- evaluation
-- compare whether the absolute value of the network output is less equal than eps
local mask = torch.le(torch.abs(mlpOutput),mlpOutput:clone():fill(eps))
if torch.sum(mask) > 0 then
  gnuplot.epsfigure('Separation surface.eps')
  local x1 = x:narrow(2,1,1) -- reshaping first input dimension for x axis
  local x2 = x:narrow(2,2,1) -- reshaping secind input dimension for y axis
  -- plotting of the collection of points that match the mask
  gnuplot.plot(x1[mask], x2[mask], '+')
  gnuplot.title('Separation surface')
```

```
    gnuplot.grid(true)
16  gnuplot.plotflush() -- save the image
    gnuplot.figure()
18 end
```

Listing 9: Drawing separation surfaces in torch

3 TensorFlow

3.1 Introduction

TensorFlow [5] is an open source software library for numerical computation and is the youngest with respect to the others Machine Learning frameworks. It was originally developed by researchers and engineers from the Google Brain Team, with the purpose of encourage research on deep architectures. Nevertheless, the environment provides a large set of tools suitable for several domains of numerical programming. The computation is conceived under the concept of *Data Flow Graphs*. Nodes in the graph represent mathematical operations, while the graph edges represent tensors (multidimensional data arrays). The core of the package is written in C++, but provides a well documented Python API. The main characteristic is its symbolic approach, which allows a general definition of a forward models, leaving the computation of the correspondent derivatives entirely to the environment itself.

3.2 Getting started

3.2.1 Python

A TensorFlow model can be easily written using Python, a very intuitive object-oriented programming language. Python is distributed with an open-source license for commercial use too. It offers a nice integration with many other programming languages and provides an extended standard library which includes `numpy` (modules designed for matrix operations, very similar to the Matlab syntax). Python runs on Windows, Linux/Unix, Mac OS X and other operative systems.

3.2.2 TensorFlow environment

Assuming that the reader is familiar with Python, here we present the building blocks of TensorFlow framework:

The Data Flow Graph To leverage the parallel computational power of multi-core CPU, GPU and even clusters of GPUs, the dynamic of the numerical computations has been conceived as a directed graph, where each node represents a mathematical operation and the edges describe the input/output relation between nodes.

Tensor It is a typed n-dimensional array that flows through the Data Flow Graph.

Variable Symbolic objects designed to represent parameters. They are exploited to compute the derivatives at a symbolical level, but in general must be explicitly initialized in a session.

Optimizer It is the component which provides methods to compute gradients from the loss function and to apply back-propagation through all the variables. A collection is available in TensorFlow to implement classic optimization algorithms.

Session A graph must be launched in a Session, which places the graph onto CPU or GPU and provides methods to run computation.

3.2.3 Installation

Information about download and installation of Python and TensorFlow are available in the official webpages[13]. Notice that a dedicated procedure must be followed for GPU installation. It's worth a quick remark on the CUDA® versions. Indeed, versions from 7.0 are officially supported, but the installation could be not straightforward in versions preceding the 8.0. Moreover, a registration to the *Accelerate Computing Developer Program*[14] is required to install the package `cuDNN`, which is mandatory to enable GPU support.

3.3 Setting up the XOR experiment

As in the previous sections of this tutorial, we show how to start managing the TensorFlow framework by facing the simple XOR classification problem by a standard ANN.

Import tensor flow

At the beginning, as for every Python library, we need to import the TensorFlow package by:

```
import tensorflow as tf

```

ciao

Dataset definition

Again, data can be defined as two matrices containing the input data and its correspondent target, called X and Y respectively. Data can be defined as a list or `numpy` array. After they will be used to fill the placeholder that actually define a type and dimensionality.

```
X = [[0,0],[0,1],[1,0],[1,1]]
Y = [[0],[1],[1],[0]]

```

Placeholders

TensorFlow provides Placeholders which are symbolic variables representing data during the computation. A Placeholders object have to be initialized with given type and dimensionality, suitable to represent the desired element. In this case we define two object `x_` and `y_` respectively for input data and target:

```
x_ = tf.placeholder(tf.float32, shape=[4,2])
y_ = tf.placeholder(tf.float32, shape=[4,1])

```

[13] Python webpage: `https://www.python.org/`, TensorFlow webpage: `https://www.tensorflow.org/`
[14] `https://developer.nvidia.com/accelerated-computing-developer`

Model definition

The description of the network depends essentially on its architecture and parameters (weights and biases). Since the parameters have to be estimated, they are defined as the variabile of the model, whereas the architecture is determined by the configuration of symbolic operations. For a 2-layers ANN we can define:

```
# Hidden units
HU = 3
# 1st layer
W1 = tf.Variable(tf.random_uniform([2,HU], -1.0, 1.0)) # weights matrix
b1 = tf.Variable(tf.zeros([HU]))   # bias
O = tf.nn.sigmoid(tf.matmul(x_, W1) + b1) # non-linear activation output
# 2nd layer
W2 = tf.Variable(tf.random_uniform([HU,1], -1.0, 1.0))
b2 = tf.Variable(tf.zeros([1]))
y = tf.nn.sigmoid(tf.matmul(O, W2) + b2)
```

The `matmul()` function performs tensors multiplication. `Variable()` is the constructor of the class variable. It needs an initialization value which must be a tensor. The function `random_uniform()` returns a tensor of a specified shape, filled with valued picked from a uniform distribution between two specified values. The `nn` module contains the most common activation functions, taking as input a tensor and evaluating the non-linear transferring component-wise (the *Logistic Sigmoid* is chosen in the reported example by `tf.nn.sigmoid()`).

Loss and optimizer definition

The cost function and the optimizer are defined by the following two lines

```
 # quadratic loss function
cost = tf.reduce_sum(tf.square(y_ - y), reduction_indices=[0])
# optimizing the function cost by gradient descent with learning step 0.1
train_step = tf.train.GradientDescentOptimizer(0.1).minimize(cost)
```

TensorFlow provides functions to perform common operations between tensors. The function `reduce_sum()` for example reduces the tensor to one (or more) dimension, by summing up along the specified dimension. The `train` module provide the most common optimizers, which will be employed during the training process. The previous code chose the *Gradient Descent* algorithm to optimize the network parameters, with respect to the penalty function defined in `cost` by using a *learning rate* equal to 0.1.

Start the session

At this point the variables are still not initialized. The whole graph exist at a symbolic level, but it is instantiated when creating a session. For example, placeholders are fed with the assigned elements in this moment.

```
% Create session
sess = tf.Session()
```

```
% Initialize variables
sess.run(tf.initialize_all_variables())
```

More specifically, `initialize_all_variables()` creates an operation (a node in the Data Flow Graph) running variables initializer. The function `Session()` creates an instance of the class *session*, while the correspondent method `run()` moves for the first time the Data Flow Graph on CPU/GPU, allocates variables and fills them with the initial values.

Training

The training phase can be defined in a `for` loop where each iteration represent a single gradient descend epoch. In the following code, some printing on the training information are added each 100 epochs.

```
Epochs = 5000 # Number of iterations

for i in range(Epochs):
  sess.run(train_step, feed_dict={x_: X, y_: Y}) # optimizer step
  if i % 100 == 0:
    print('Epoch ', i)
    print('Cost ', sess.run(cost, feed_dict={x_: X, y_: Y}))

```

The `sess.run()` calling runs the operations previously defined for the first argument, which in this case is an optimizer step (defined by `train_step`). The second (optional) argument for `run` is a dictionary `feed_dict`, pairing each placeholder with the correspondent input. The function `run()` is used also to evaluate the cost each 100 epochs.

Evaluation

The performance of the trained model can be easily evaluated by:

```
correct_prediction = abs(y_ - y) < 0.5
cast = tf.cast(correct_prediction, "float")
accuracy = tf.reduce_mean(cast)

yy, aa = sess.run([y, accuracy],feed_dict={x_: X, y_: Y})

print "Output: ", yy
print "Accuracy: ", aa

```

Draw separation surfaces

In order to visualize separation surfaces computed by the network, it can be useful to generate a random sample of points on which test results, as showed in Figure 3.

```
plt.figure()
# Plotting dataset
c1 = plt.scatter([1,0], [0,1], marker='s', color='gray', s=100)
c0 = plt.scatter([1,0], [1,0], marker='^', color='gray', s=100)
# Generating points in [-1,2]x[-1,2]
DATA_x = (np.random.rand(10**6,2)*3)-1
DATA_y = sess.run(y,feed_dict={x_: DATA_x})
# Selecting borderline predictions
ind = np.where(np.logical_and(0.49 < DATA_y, DATA_y< 0.51))[0]
DATA_ind = DATA_x[ind]
# Plotting separation surfaces
ss = plt.scatter(DATA_ind[:,0], DATA_ind[:,1], marker='_', color='black', s=5)
# Some figure's settings
plt.legend((c1, c0, ss), ('Class 1', 'Class 0', 'Separation surfaces'), scatterpoints=1)
plt.xlabel('Input x1')
plt.ylabel('Input x2')
plt.axis([-1,2,-1,2])
plt.show()
```

Listing 10: Draw separation surfaces

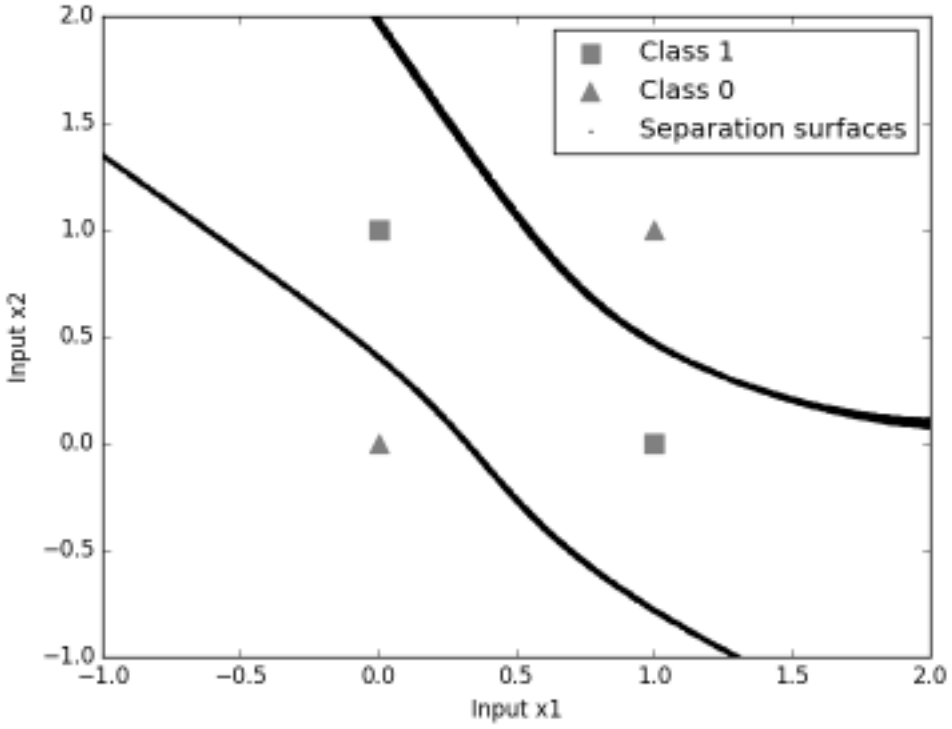


Figure 3: Separation surfaces on the XOR classification task obtained by 2-layer ANN with 3 Hidden Units and the *Logistic Sigmoid* as activation and output function.

4 MNIST Handwritten Characters Recognition

In this Section we show how to set up a 2-Layer ANN in order to face the MNIST [6] classification problem, a well known data set for handwritten characters recognition. It is extensively used to test and compare general Machine Learning algorithms and Computer Vision methods. Data are provided as 28×28 pixels (grayscale) images of handwritten digits. The training and test sets contain respectively 60,000 and 10,000 instances. Files .zip are available at the official site[15], together with a list of performance achieved by most common algorithms. We show the setting up of a standard 2-Layer ANN with 300 units in the hidden layer, represented in Figure 4, since it is one of the architecture reported in the official website and the obtained results can be easily compared. The input will be reshaped so as to feed the network with a 1-Dimensional vector with $28 \cdot 28 = 784$ elements. Each image is originally represented by a matrix containing the grayscale value of the pixels in $[0, 255]$, which will be normalized in $[0, 1]$. The output will be a 10 elements prediction vector, since labels for each element will be expressed by the one-hot encoding binary vector of 10 null bits, with only a 1 in the position indicating the class. Activation and penalty functions are different within different environments to provide an overview on different approaches.

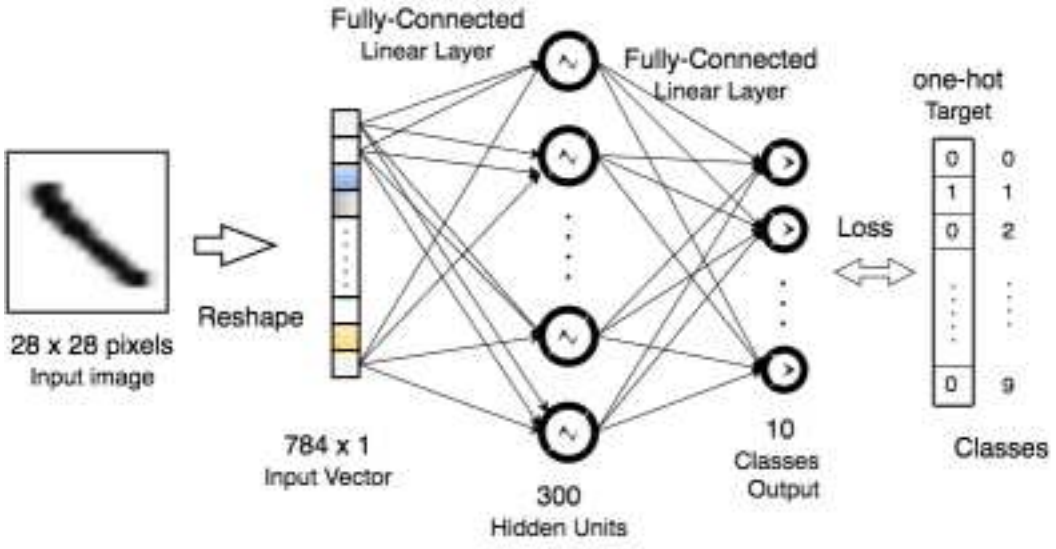


Figure 4: General architecture of a 2-Layer network model proposed to face the MNIST data.

4.1 MNIST on MATLAB

Once data have been downloaded from the official MNIST site, we can use MATLABfunctions available at the Stanford University site[16] to extract data from files and organize them in *inputSize* –by–*numberOfSamples* matrix form. The extraction routines reshape (so as that each digit is represented by a 1-D column vector of size 784) and normalizes data (so as that each feature lies in the interval [0, 1]) . Once unzipped data and functions in the same folder, it is straightforward to upload images in the MATLABworkspace by the `loadMNISTImages` function:

[15] `http://yann.lecun.com/exdb/mnist/`.

[16] `http://ufldl.stanford.edu/wiki/index.php/Using_the_MNIST_Dataset`.

```
images_Train = loadMNISTImages('train-images.idx3-ubyte');
images_Test = loadMNISTImages('t10k-images.idx3-ubyte');
images = [images_Train, images_Test];
```

where training and test set have been grouped in the same matrix to evaluate performance on the provided test set during the training. Correspondent labels can be loaded and grouped in a similar way by the function `loadMNISTLabels`:

```
labels_Train = loadMNISTLabels('train-labels.idx1-ubyte');
labels_Test = loadMNISTLabels('t10k-labels.idx1-ubyte');
labels = [labels_Train; labels_Test];
```

The original labels are provided as a 1-Dimensional vector containing a number from 0 to 9 according to the correspondent digit. The one-hot encoding target matrix for the whole dataset can be generated exploiting the MATLABfunction `ind2vec`[17]:

```
labels = full( ind2vec( labels' + 1 ) );
```

To check the obtained results, we replied one of the 2-layer architectures listed at the official website, which is supposed to reach around 96% of accuracy with 300 hidden units and can be initialized by:

```
nn = patternnet( 300 );
```

As already said, this command creates a 2-Layer ANN where the hidden layer has 300 units and the *Hyperbolic Tangent* as activation, whereas the output function is computed by the *softmax*. The (default) penalty function is the *Cross-Entropy Criterion*.

In this case we change the data splitting so as that data used for test comes only from the original test set (which has been concatenated with the training one), prevent to mix samples among Train, Validation and Test set. This step is completely customizable by the method `divideFcn` and the fields of the options `divideParam`. The `divideind` method picks data according to the provided indexes for the data matrix:

```
nn.divideFcn = 'divideind';
nn.divideParam.trainInd = 1:45000;
nn.divideParam.valInd = 45000:60000;
nn.divideParam.testInd = 60000:70000;
```

In this case, we arbitrarily decided to use the last 25% of the Training data for Validation, since the samples are more or less equally distributed by classes.

As already said, network training can be started by:

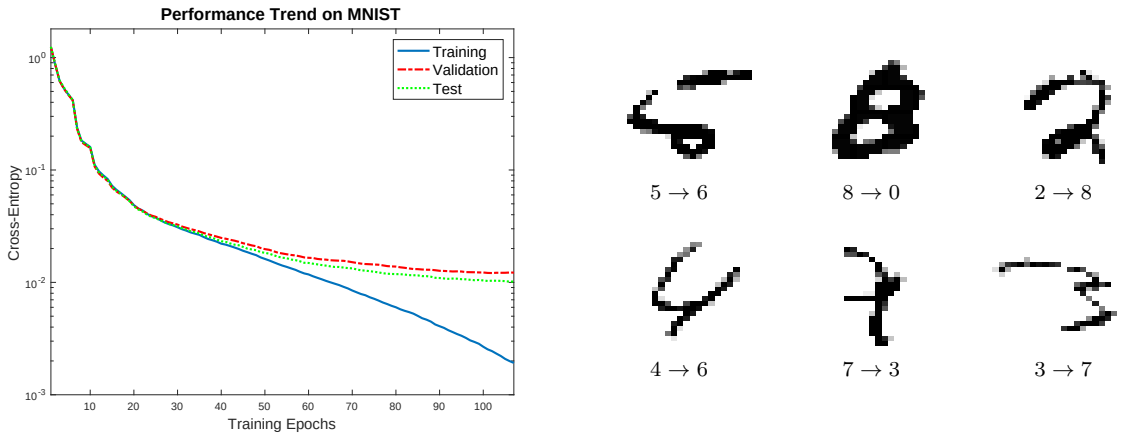
[17]The function `full` prevent for MATLABautomatically convert to sparse matrix, which in our tests may cause some problems at the calling of the function `train`.

```
[ nn, tr ] = train( nn, images, labels );
```

In the reported case, the training stopped after 107 epochs because of an increasing in the validation error (see Section 1.3). The performance during training are shown in Figure 5(a), which is obtained by the following code:

```
%%%% Plotting MNIST training performance

plot(tr.perf,'LineWidth',2); % plot training error
hold on;
plot(tr.vperf,'r-.','LineWidth',2); % plot validation error
plot(tr.tperf,'g:','LineWidth',2); % plot test error
set(gca,'yscale','log'); % setting log scale
axis([1,107,0.001,1.8]);
xlabel('Training Epochs','FontSize',14);
ylabel('Cross-Entropy','FontSize',14);
title('Performance Trend on MNIST','FontSize',16);
h = legend({'Training','Validation','Test'},'Location','NorthEast');
set(h ,'FontSize',14);
```



(a) (b)

Figure 5: On the left the performance on the MNIST dataset during the training of a 2-layer ANN with 300 hidden units. Training is stopped after 107 epochs for validation checking. On the right we report some misclassified samples. The network reaches about 96% of classification accuracy on the test set (in accordance with the ones provided at the MNIST web page).

In Figure 5(b) we show some misclassified digits, indicating the original label and the predicted one. The visualization is obtained by the Matlab function `image` (after a reshaping to the original square dimensions and grayscale, multiplying by 255). In Listing 11 we show how to evaluate classification accuracy and confusion matrix on data, which should give coherent results with respect to which reported in the official site for the same architecture (about 4% error on test set).

```
% network evaluation
f = nn( images( :, 1:45000 ) ); % training predictions
fv = nn( images( :, 45001:60000 ) ); % validation predictions
ft = nn( images( :, 60001:end ) ); % test predictions
% classification accuracy
A = mean( vec2ind(f) == vec2ind( labels( :, 1:45000 ) ) );
Av = mean( vec2ind(fv) == vec2ind( labels( :, 45001:60000 ) ) );
At = mean( vec2ind(ft) == vec2ind( labels( :, 60001:end ) ) );
% confusion matrix
C = confusionmat(  vec2ind(f), vec2ind( labels( :, 1:45000 ) ) ) ;
Cv = confusionmat( vec2ind(fv), vec2ind( labels( :, 45001:60000 ) ) );
Ct = confusionmat( vec2ind(ft), vec2ind( labels( :, 60001:end ) ) );
```

Listing 11: Performance evaluation of the network trained on the MNIST

4.2 MNIST on Torch

As already said, the Torch environment provides a lot of tools for Machine Learning, included a lot of routines to download and prepare most common Datasets. A wide overview on most useful tutorials, demos and introduction to most common methods can be found in a dedicate webpage[18], including a *Loading popular datasets* section. Here, a link to the MNIST loader page [19] is available, where data and all the informations for the correspondent `mnist` package installation are provided. After the installation, data can be loaded by the code in Listing 12.

```

% loading data
images_Train = loadMNISTImages('train-images-idx3-ubyte');
images_Test = loadMNISTImages('t10k-images-idx3-ubyte');
images = [ images_Train, images_Test];

% loading targets
labels_Train = loadMNISTLabels('train-labels-idx1-ubyte');
labels_Test = loadMNISTLabels('t10k-labels-idx1-ubyte');
labels = [ labels_Train; labels_Test];
labels = full(  ind2vec( labels' + 1 ) );
%%
% initialization
nn = patternnet( 300 );
nn.divideFcn = 'divideind';
nn.divideParam.trainInd = 1 : 45000 ;
nn.divideParam.valInd = 45001 : 60000 ;
nn.divideParam.testInd = 60001 : 70000 ;
```

Listing 12: Loading MNIST data

Data will be loaded as a table named *train*, where digits are expressed as a *numberOfSamples*-by-28-by-28 tensor of type `ByteTensor` stored in the field *data*, expressing the value of the gray levels of each pixel between 0 and 255. Targets will be stored as a 1-D vector, expressing the digits labels, in the field *label*. We have to convert data to the `DoubleTensor` format and, again, normalize the input features to have values in $[0, 1]$ and reshape the original 3-D tensor in a 1-D input vector.

[18] `https://github.com/torch/torch7/wiki/Cheatsheet#machine-learning`
[19] `https://github.com/andresy/mnist`

Labels have to be incremented by 1, since `CrossEntropyCriterion` accepts target indicating the class avoiding null values (i.e. 1 means a sample to belong to the first class and so on). In the last row we perform a random shuffling of data in order to prepare the train/validation splitting by the function:

```
function rndShuffle(dataset)
  -- random data shuffle
  local n = dataset.data:size(1)
  local perm = torch.LongTensor():randperm(n)
  dataset.data = dataset.data:index(1, perm)
  dataset.label = dataset.label:index(1, perm)
  return dataset
end
```

At this point we can create a *validation* set from the last quarter of the training data by:

```
-- splitting function definition
function splitData(data, p)
  -- splits data depending on the rate p
  local p = p > 0 and p <= 1 and p or 0.9
  local trainSize = torch.floor(p*data:size(1))
  return data:narrow(1,1,trainSize),data:narrow(1,1 + trainSize, data:size(1) -
    trainSize)
end

validation = {}
trainRate = 0.75
train.data, validation.data = splitData(train.data, trainRate)
train.label, validation.label = splitData(train.label, trainRate)
```

The code to build the proposed 2-layer ANN model is reported in Listing 13, where the network is assembled in the `Sequential` container, using this time the *ReLU* as activation for the hidden layer, whereas output and penalty functions are the same used in the previous section (*softmax* and *Cross-Entropy* respectively).

```
-- mlpwork definition
require 'nn'

local mlp = nn.Sequential()
mlp:add(nn.Linear(784, 300))
mlp:add(nn.ReLU())
mlp:add(nn.Linear(300, 10))
mlp:add(nn.SoftMax())

-- loss function
local c = nn.CrossEntropyCriterion()
```

Listing 13: Network definition for MNIST data

The network training can be defined in a way similar to the one proposed in Listing 6. Because of the width of the training data, this time is more convenient to set up a *minibatch* training as showed in Listing 14. Moreover, we also define an early stopping criterion which stops the training when

the penalty on the validation set start to increase, preventing overfitting problems. The training function expects as inputs the network (named *mlp*), the criterion to evaluate the loss (named *criterion*), training and validation data (named *trainset* and *validation* respectively) organized as a table with fields *data* and *label* as defined in Listing 12. An optional configuration table *options* can be provided, indicating the number of training epochs (*nepochs*), the learning rate (*learning_rate*), the mini-batch size (*batchSize*) and the number of consecutive increasings in the validation loss which causes a preventive training stop (*maxSteps*). It is worth a remark on the function `split`, defined for the `Tensor` class, used to divide data in batches stored in an indexed table. At the end of the training, a vector containing the loss evaluated at each epoch is returned. The validation loss is computed with the help of the function `evaluate`, which splits again the computation in smaller batches, preventing from too heavy computations when the number of parameters and samples is very large.

```
function training(mlp, criterion, trainset, validation, options)
-- minibatch training
assert(mlp and trainset, "At least 2 arguments are expected")
local nepochs = options and options.nepochs or 1000 -- max number of epoch
local learning_rate = options and options.learning_rate or 0.01
local batchSize = options and options.batchSize or 32
local maxSteps = options and options.maxSteps or 10 -- max validation fails
local input, target = trainset.data, trainset.label
-- vector for saving loss during epoch
local loss = torch.Tensor(nepochs):typeAs(input):fill(0)
local valLoss = torch.Tensor(nepochs):typeAs(input):fill(0)
-- minibatches splitting
local minibatches, minitarget = input:split(batchSize), target:split(batchSize)
-- last loss and validation fails for early stopping criterion
-- based on validation loss
local lastLoss,step = 0,-1
local epoch = 1
while epoch < nepochs and step < maxSteps do
for bi, batch in pairs(minibatches) do
loss[epoch] = loss[epoch] + criterion:forward(mlp:forward(batch), minitarget[bi])
-- reset gradients to null values
mlp:zeroGradParameters()
-- accumulate gradients
mlp:backward(batch, criterion:backward(mlp.output, minitarget[bi]))
-- update parameters
mlp:updateParameters(learning_rate)
if bi%100 == 0 then collectgarbage() end -- cleaning nil variable
end
-- evaluating validation loss
local currLoss = evaluate(mlp, criterion, validation)
valLoss[epoch] = currLoss
if currLoss >= lastLoss*0.9999 then
step = step + 1
else
step = 0
end
lastLoss = currLoss
xlua.progress(epoch, nepochs) -- printing epochs progress
epoch = epoch + 1
end
if step == maxStep then
```

```
print('Training stopped at Epoch: '..epoch..
' because of Validation Error increasing in the last '
..maxStep..' epochs')
end
return loss:narrow(1,1,epoch-1)/#minibatches, valLoss:narrow(1,1,epoch-1)
end

function evaluate(mlp, criterion, dataset, options)
-- evaluate the loss from criterion between mpl predictions
-- by accumulating within minibatches from dataset
assert(mlp and dataset, "At least 2 arguments are expected")
local batchSize = options and options.batchSize or 32
local input, target = dataset.data, dataset.label
local loss = 0
local minibatches, minitarget = input:split(batchSize), target:split(batchSize)
for bi, batch in pairs(minibatches) do
loss = loss + criterion:forward(mlp:forward(batch), minitarget[bi])
end
mlp:zeroGradParameters()
return loss/#minibatches
end
```

Listing 14: Sample of a mini-batch training function

In Listing 15 we show how to compute the Confusion Matrix and the Classification Accuracy on data by the function `confusionMtx`, taking in input the network (*mlp*), data (*dataset*) and the expected number of classes (*nclasses*).

```
function confusionMtx(mlp, dataset, nclasses)
local input, target = dataset.data, dataset.label
local confMtx = torch.zeros(nclasses,nclasses):typeAs(input)

local prediction = mlp:forward(input)
-- get the position of the unit with the maximum value
local _, pos = torch.max(prediction,2)
local acc = 0
for i = 1,nclasses do
local predicted = torch.eq(pos,i):typeAs(input)
for j = 1,nclasses do
local truth =  torch.eq(target, j):typeAs(input)
confMtx[i][j] = torch.cmul(predicted, truth):sum(1)
if i == j then acc = acc + confMtx[i][j] end
end
end
return confMtx, acc/input:size(1)
end
```

Listing 15: Evaluating Confusion Matrix and Accuracy

At this point we can start a trial by the following code:

```
options = {nepochs = 250, batchSize = 64, learning_rate = 0.05, maxStep = 50}
training(mlp, c, train, validation, options)
confMtx, acc = confusionMtx(mlp, test, 10)
```

In this case the training is stopped by the validation criterion after epoch 117, producing a Classification Accuracy on test of about 97%. In Figure 6(a) we report the trend of the error during training. Since in general can be useful to visualize the confusion matrix (which in this case is almost diagonal), in Figure 6(b) we show the one obtained by the function `imagesc` from the package `gnuplot`, which just give a color map of the matrix passed as input.

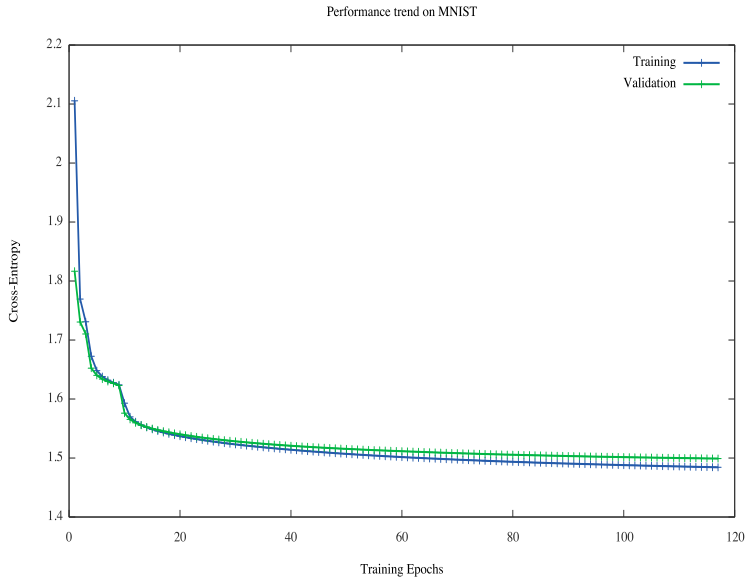


Figure 6: Confusion matrix on the MNIST test set.

4.3 MNIST on Tensor Flow

Even TensorFlow environment makes available many tutorials and preloaded dataset, including the MNIST. A very fast introductive section for beginners can be found at the official web page[20]. However, we will show some functions to download and import the dataset in a very simple way. Indeed, data can be directly loaded by:

```
from tensorflow.examples.tutorials.mnist import input_data
mnist = input_data.read_data_sets("MNIST_data/", one_hot=True)
```

We firstly define two auxiliary functions. The first (`init_weights`) will be used to initialize the parameters of the model, whereas the second (`mlp_output`) to compute the predictions of the model.

```
def init_weights(shape):
return tf.Variable(tf.random_uniform(shape, -0.1, 0.1))

def mlp_output(X, W_h, W_o, b_h, b_o):

a1 = tf.matmul(X, W_h) + b_h
o1 = tf.nn.relu(a1)  #output layer1

```

[20] `https://www.tensorflow.org/versions/r0.12/tutorials/mnist/beginners/index.html`

```
a2 = tf.matmul(o1, W_o) + b_o
o2 = tf.nn.softmax(a2)  #output layer2

return o2
```

Now, with the help of the proposed function `init_weights`, we define the parameters to be learned during the computation. $W1$ and $b1$ represent respectively the weights and the biases of the hidden layer. Similarly, $W2$ and $b2$ are respectively the weights and the biases of the output layer.

```
W1 = init_weights([x_dim, h_layer_dim])
b1 = init_weights([h_layer_dim])
W2 = init_weights([h_layer_dim, y_dim])
b2 = init_weights([y_dim])
```

Once we defined the weights, we can symbolically compose our model by the calling of our function `mlp_output`. As in the XOR case, we have to define a placeholder storing the input samples.

```
# Input
x = tf.placeholder(tf.float32, [None, x_dim])
# Prediction
y = mlp_output(x, W1, W2, b1, b2)
```

Then we need to define a cost function and an optimizer. However, this time we add the square of the Euclidean Norm as regularizer, and the global cost function is composed by the *Cross-Entropy* plus the regularization term scaled by a coefficient of 10^{-4}. At the beginning of the session, TensorFlow moves the Data Flow Graph to the CPUs or GPUs and initializes all variables.

```
# Model Specifications
LEARNING_RATE = 0.5
EPOCHS = 5000
MINI_BATCH_SIZE = 50
# Symbolic variable for the target
y_ = tf.placeholder(tf.float32, [None, y_dim])
# Loss function and optimizer
cross_entropy = tf.reduce_mean(tf.nn.softmax_cross_entropy_with_logits(y, y_))
regularization = (tf.reduce_sum(tf.square(W1), [0, 1])
                  + tf.reduce_sum(tf.square(W2), [0, 1]) )
cost = cross_entropy + 10**-4 * regularization
train_step = tf.train.GradientDescentOptimizer(LEARNING_RATE).minimize(cost)
# Start session and initialization
sess = tf.Session()
sess.run(tf.initialize_all_variables())
```

TensorFlow provides the function `next_batch` for the `mnist` class to randomly extract batches of a specified size. Data are split in shuffled partitions of the indicated size and, by means of an implicit counter, the function slide along batches at each calling allowing a fast implementation for mini-batch Gradient Descent method. In our case, we used a `for` loop to scan across batches, executing a training step on the extracted data at each iteration. Once the whole Training set

has been processed, the loss on Training, Validation and Test sets is computed. These operations are repeated in a `while` loop, whose steps representing the epochs of training. The loop stops when the maximum number of epochs is reached or the network start to overfit Training data. The early stopping is implemented by checking the Validation error and training is stopped when no improvements are obtained for a fixed number of consecutive epochs (`val_max_steps`). The maximum number of epochs and learning rate must be set in advance.

```
# Save values to be plotted
errors_train=[]
errors_test=[]
errors_val=[]

# Early stopping (init variables)
prec_err = 10**6 # just a very big value
val_count = 0
val_max_steps = 5

# Training
BATCH_SIZE = np.shape(mnist.train.images)[0]
MINI_BATCH_SIZE = 50

i = 1
while i <= epochs and val_count < val_max_steps:

  # Train over the full batch is performed with mini-batches algorithm
  for j in range(BATCH_SIZE/MINI_BATCH_SIZE):
    batch_xs, batch_ys = mnist.train.next_batch(50)
    sess.run(train_step, feed_dict={x: batch_xs, y_: batch_ys})

  # Compute error on validation set and control for early-stopping
  curr_err = sess.run(cross_entropy,
        feed_dict={x: mnist.validation.images, y_: mnist.validation.labels})
  if curr_err >= prec_err*0.9999:
    val_count = val_count + 1
  else:
    val_count = 0

  prec_err = curr_err

  # Save values for plot
  errors_val.append(curr_err)
  c_test = sess.run(cross_entropy,
        feed_dict={x: mnist.test.images, y_: mnist.test.labels})
  errors_test.append(c_test)
  c_train = sess.run(cross_entropy,
        feed_dict={x: mnist.train.images, y_: mnist.train.labels})
  errors_train.append(c_train)

  # Print info about the current epoch
  print "\n\nEPOCH: ",i, "/", epochs
  print "  TRAIN ERR: ", c_train
  print "  VALIDATION ERR: ", curr_err
  print "  TEST ERR: ", c_test
  print "\n(Early stopping criterion: ", val_count, "/", val_max_steps, ")"
```

```
    # Increment epochs-index
    i = i+1
```

The prediction accuracy of the trained model can be evaluated over the test set in way similar to the one presented for the XOR problem. This time we need to exploit the *argmax* function to convert the one-hot encoding in the correspondent labels.

```
# Symbolic formulas
correct_prediction = tf.equal(tf.argmax(y,1), tf.argmax(y_,1))
accuracy = tf.reduce_mean(tf.cast(correct_prediction, tf.float32))
# Compute accuracy on the test set
aa = sess.run(accuracy, feed_dict={x: mnist.test.images, y_: mnist.test.labels})
print "Accuracy: ", aa
```

The trend of the network performance showed in Figure 7 during training can be obtained by the following code:

```
E = range(np.shape(errors_train)[0])

line_train, = plt.plot(E, errors_train)
line_test, = plt.plot(E, errors_test)
line_val, = plt.plot(E, error_val)
plt.legend([line_train, line_val, line_test], ['Training', 'Validation', 'Test'])
plt.ylabel('Cross-Entropy')
plt.xlabel('Epochs')
plt.show()
```

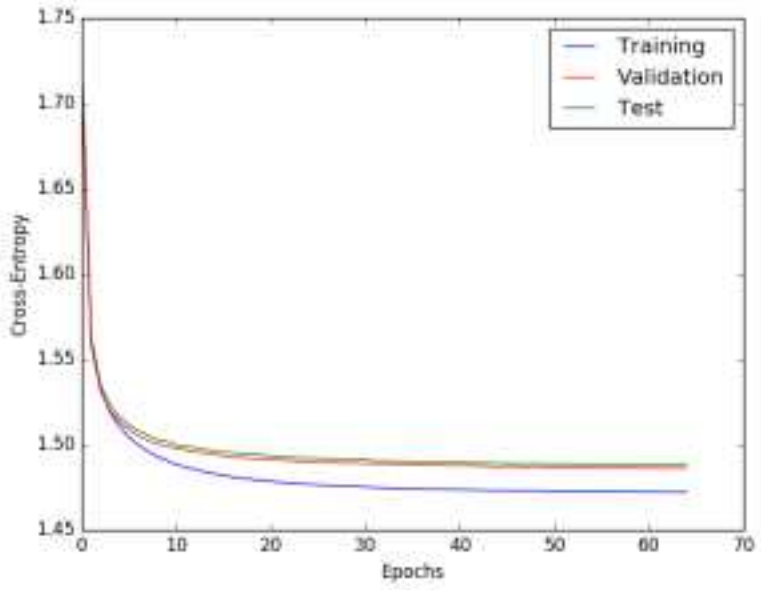


Figure 7: Error trend on the MNIST dataset during the training of a 2-layer ANN with 300 hidden units.

5 Convolutional Neural Networks

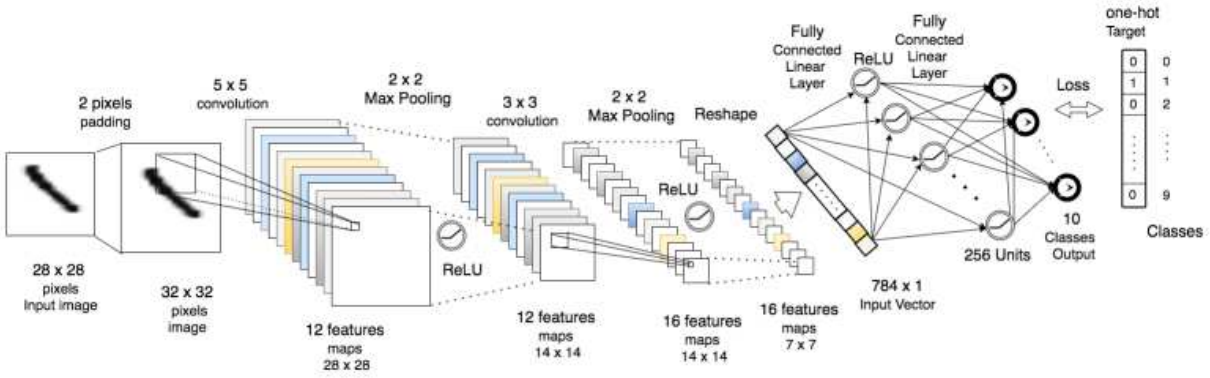


Figure 8: General architecture of the CNN model proposed to face the MNIST data.

In this section we introduce the Convolutional Neural Networks (CNNs [7, 6, 8]), an important and powerful kind of learning architecture widely diffused especially for Computer Vision applications. They currently represent state of the art algorithm for image classification tasks and constitute the main architecture used in Deep Learning. We show how to build and train such a structure within all the proposed frameworks, exploring the most general functions and setting up few experiments on MNIST pointing out some important features.

5.1 MATLAB

Main function and classes to build and train CNNs with MATLABare contained again in Neural Network Toolbox™ and Statistic and Machine Learning Toolbox™ . Nevertheless, the Parallel Computing Toolbox™ becomes necessary too. Moreover, to train the network a CUDA® -enabled NVIDIA® GPU is required. Again, we do not focus too much on main theoretical properties and general issues about CNNs but only on main implementation instruments.
The network has to be stored in an MATLABobject of kind `Layer`, which can be sequentially composed by different sub-layers. For each one, we do not list all the available options, which as always can be explored by the interactive help options from the Command Window[21]. Most common convolutional objects can be defined by the functions:

`imageInputLayer` creates the layer which deals with the original input image, requiring as argument a vector expressing the size of the input image given by *height* by *width* by *number of channels*;

`convolution2dLayer` defines a layer of 2-D convolutional filters whose size is specified by the first argument (a real number for a square filter, a 2-D vector to specify both height and width), whereas the second argument specifies the total number of filters; main options are `'Stride'`, which indicates the sliding step (default `[1, 1]` means 1 pixel in both directions), and `'Padding'` (default `[0, 0]`), whose have to appear in Name,Value pairs;

`reluLayer` defines a layer computing the Rectifier Activation Linear Unit (ReLU) for the filter outputs;

[21] Further documentation is available at the official site `https://it.mathworks.com/help/nnet/convolutional-neural-networks.ht`

`averagePooling2dLayer` layer computing a spatial reduction of the input by averaging the values of the input on each grid of given dimension (default `[2, 2]`, `'Stride'` and `'Padding'` are options too);

`maxPooling2dLayer` layer computing a spatial reduction of the input assigning the max value to each grid of given dimensions (default `[2, 2]`, `'Stride'` and `'Padding'` are options too);

`fullyConnectedLayer` requires the desired output dimension as argument and instantiates a classic fully connected linear layer, the number of the connections is adapted to fit the input size;

`dropoutLayer` executes a dropout units selection with the probability given as argument;

`softmaxLayer` computes a probability normalization based on the *softmax* function;

`classificationLayer` adds the final classification layer evaluating the *Cross-Entropy* loss function between predictions and labels

```
CnnM = [ imageInputLayer([28 28 1]), ... % input layer
... % convolutional layer 12 filters of size 5x5 and 2 pixels 0-padding
convolution2dLayer(5,12,'Padding',2), ...
reluLayer(), ...
... % max-pooling layer on 2x2 grid and 2 pixels step in both directions
maxPooling2dLayer(2,'Stride',2), ...
... % convolutional layer with 16 filters of size 3x3 and 1 pixels 0-padding
convolution2dLayer(3,16,'Padding',1), ...
reluLayer(), ...
... % max-pooling layer on 2x2 grid and 2 pixels step in both directions
maxPooling2dLayer(2,'Stride',2), ...
... % classic fully connected layer with 256 output
fullyConnectedLayer(256), ...
reluLayer(), ...
... % classic fully connected layer with 10 output
fullyConnectedLayer(10), ...
softmaxLayer(), ...
classificationLayer() ];
```

Listing 16: Definition of a CNN to face the MNIST within Matlab framework.

In Listing 16 we show how to set up a basic CNN to face the 28×28 pixels images from MNIST showed in Figure 8. The global structure of the network is defined as a vector composed by the different modules. The initialized object `Layer`, named *CnnM*, can be visualized from the command window giving:

```
>> CnnM

CnnM =

  12x12 Layer array with layers:

     1   ''   Image Input             28x28x1 images with 'zerocenter' normalization
     2   ''   Convolution             12 5x5 convolutions with stride [1  1] and padding [2  2]
     3   ''   ReLU                    ReLU
     4   ''   Max Pooling             2x2 max pooling with stride [2  2] and padding [0  0]
     5   ''   Convolution             16 3x3 convolutions with stride [1  1] and padding [1  1]
     6   ''   ReLU                    ReLU
     7   ''   Max Pooling             2x2 max pooling with stride [2  2] and padding [0  0]
     8   ''   Fully Connected         256 fully connected layer
     9   ''   ReLU                    ReLU
    10   ''   Fully Connected         10 fully connected layer
    11   ''   Softmax                 softmax
    12   ''   Classification Output   cross-entropy
```

Data for the MNIST can be loaded by the Stanford routines showed in Section 4.1. This time input images are required as a 4-D tensor of size *numberOfSamples*–by–*channels*–by–*height*–by–*width* and, hence, we have to modify the provided function `loadMNISTimages` or just to reshape data, as showed in the first line of the following code:

```
X = reshape(images_Train,28,28,1,[])
Y = nominal(labels_Train,{'0','1','2','3','4','5','6','7','8','9'});
```

The second command is used to convert targets in a `categorical` MATLABvariable of kind `nominal`, which is required to train the network exploiting the function `trainNetwork`. It also require as input an object specifying the training options which can be instantiated by the function `trainingOptions`. The command:

```
opts = trainingOptions('sgdm');
```

selects (by the `'sgdm'` string) the *Stochastic Gradient Descent* algorithm using momentum. Many optional parameters are available, which can be set by additional parameter in the *Name,Value* pairs notation again. The most common are:

`Momentum` (default 0.9)

`InitialLearnRate` (default 0.01)

`L2Regularization` (default 0.0001)

`MaxEpochs` (default 30)

`MiniBatchSize` (default 128)

After this configuration, the training can be started by the command:

```
[trainedNet,trainOp] = trainNetwork(X,Y,CnnM,opts)
```

where *trainedNet* will contain the trained network and *trainOp* the training variables. Training starts a command line printing of some useful variables indicating the training state, which will be similar to:

Epoch	Iteration	Time Elapsed (seconds)	Mini-batch Loss	Mini-batch Accuracy	Base Learning Rate
1	50	1.03	2.2867	46.09%	0.001000
1	100	2.03	1.4157	65.62%	0.001000
1	150	3.14	0.4497	79.69%	0.001000
1	200	4.29	0.3388	87.50%	0.001000
1	250	5.42	0.1671	95.31%	0.001000
1	300	6.52	0.1612	93.75%	0.001000
1	350	7.64	0.1120	96.88%	0.001000

At the end of the training, we can evaluate the performance on a suitable test set *Xtest*, together with its correspondent target *Ytest*, by the function `classify`:

```
Predictions = classify(trainedNet,Xtest);
Accuracy = mean(Predictions==Ytest);
C = confusionmat(P,categorical(Y));
```

Assuming *Ytest* to be a 1-Dimensional vector of class labels, classification accuracy can be calculated as before by the meaning of the boolean comparing with the computed predictions vector *Predictions*. An useful built-in function to compute the confusion matrix is provided, requiring the `nominal` labels to be converted into `categorical` as the predictions. In this setting, the final classification accuracy on the test set should be close to the 99%.

When dealing with CNNs, an important new type of object introduced are the convolutional filters. We do not want to go in deep with theoretical explanations, however sometimes it could be useful to visualize the composition of the convolutional filters in order to get an idea of which kind of features each filter detects. Filters are represented by the weights of each convolution, stored in each layers in the 4-D weights tensor of size *height*–by–*width*–by–*numberOfChannels*–by–*numberOfFilters*. In our case for example, the filters of the first convolutional be accessed by the notation `CnnM.Layer(2).Weights`. In Figure 9, we show their configuration after the training (again exploiting the function `image` and the colormap `gray` and a normalization in $[0, 255]$ for a suitable visualization).

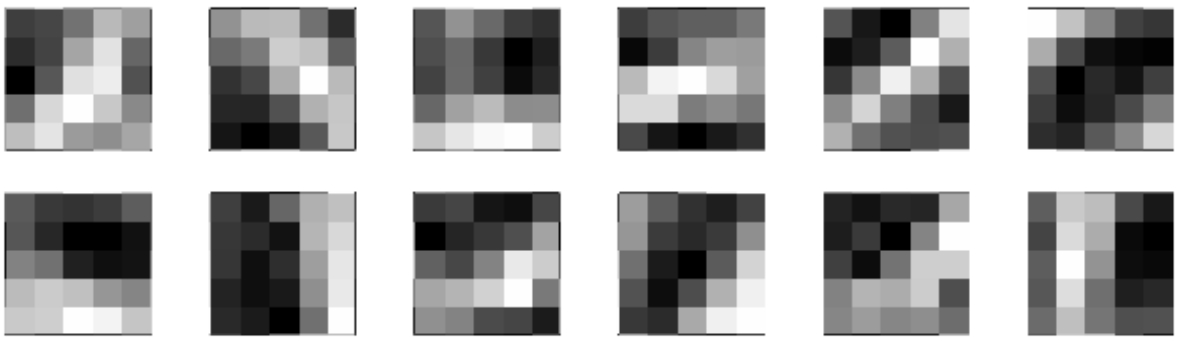


Figure 9: First Convolutional Layer filters of dimension 5×5 after the training on the MNIST images with MATLAB.

5.2 Torch

Within the Torch environment is straightforward to define a CNN from the `nn` package presented in Section 2.2.2. Indeed, the network can be stored in a container and composed by specific modules. Again, we give a short list description of the most common ones, which can be integrated with the standard transfer functions or with a standard linear (fully connected) layer introduced before:

`SpatialConvolution` defines a convolutional layers, the required arguments are the number of input channels, the number of output channels, the height and the width of the filters. The step-size and zero-padding height and width are optional parameters (with default value 1 and 0 respectively)

`SpatialMaxPooling` standard max pooling layer, requiring as inputs height and width of the pooling window, whereas the step-sizes are optional parameters (default the same as the window size)

`SpatialAveragePooling` standard average pooling layer, same features of the previous one

`SpatialDropout` set a dropout layer taking as optional argument the deactivating rate (default 0.5)

`Reshape` is a module which is usually used to unroll the output after a convolutional/pooling process as a 1-D vector to be feed to a linear layer, takes as input the size of the desired output dimensions

The assembly of the network follows from what seen until now. To have a different comparison with the previous experiment, we operate an initial 2×2 window max-pooling on the input image, in order to provide

the network by images of lower resolution. The general architecture will differ from the one defined in Section 5.1 only by the first layers. The proposed network is generated by the code in Listing 17, whereas in Figure 10 we show the global architecture.

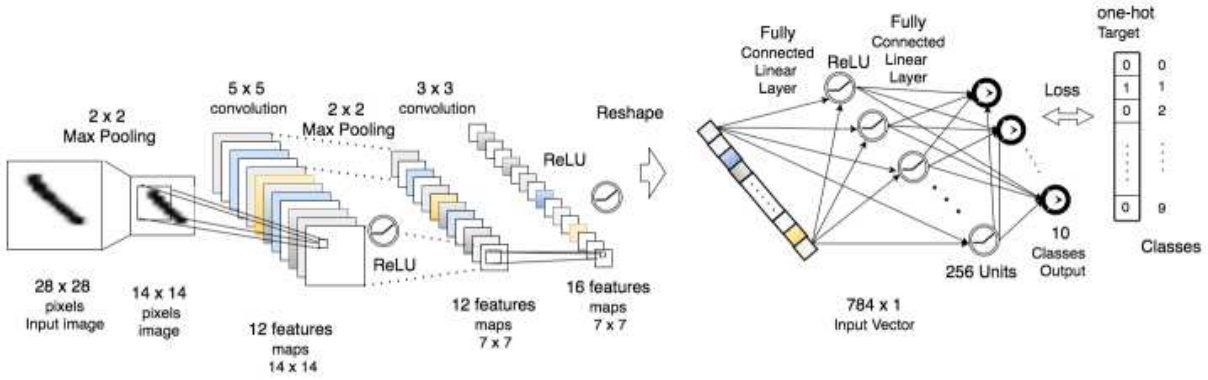


Figure 10: General architecture of the CNN model proposed to face the MNIST data within the Torch environment (Section 5.2).

```
require 'nn'
-- container definition
cnnet = nn.Sequential()
-- network assembly
cnnet:add(nn.SpatialConvolution(1,12,5,5,1,1,2,2))
cnnet:add(nn.ReLU())
cnnet:add(nn.SpatialMaxPooling(2,2,2,2))
cnnet:add(nn.SpatialConvolution(12,16,3,3,1,1,1,1))
cnnet:add(nn.ReLU())
cnnet:add(nn.Reshape(7*7*16))
cnnet:add(nn.Linear(7*7*16,256))
cnnet:add(nn.ReLU())
cnnet:add(nn.Linear(256,10))
cnnet:add(nn.SoftMax())
```

Listing 17: Network definition for MNIST data reduced to 14×14 pixels images by 2×2 max-pooling

If we use the training function defined in Listing 14, the optimization (starting with the same options) stops for validation check after 123 epochs, producing a Classification Accuracy of about 90%. This just to give an idea of the difference in the obtained performances when there is a reduction in the information expressed by input images. In Figure 11 we show the 12 filters of size 5×5 extracted by the first convolutional layer. The weights can be obtained by the function `parameters` from the package `nn`:

```
myParam = cnnet:parameters()
```

which return an indexed table storing the weights of each layer. In this case, the first element of the table contains a tensor of dimension 12 by 25 representing the weights of the filters. The visualization can be generated exploiting the function `imagesc` from the package `gnuplot`, after reshaping each line in the 5×5 format.

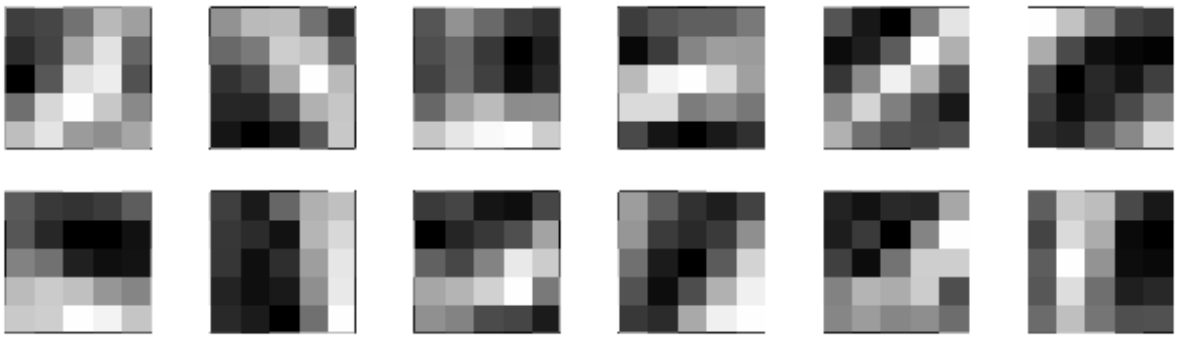


Figure 11: First Convolutional Layer filters of dimension 5×5 after the training with Torch on the MNIST images halved by 2×2 max pooling.

5.3 Tensor Flow

In this section we will show how to build a CNN to face the MNIST data using TensorFlow. The first step is to import libraries, the Mnist Dataset and to define main variables. Two additional package are required: `numpy` for matrices computations and `matplotlib.pyplot` for visualization issues.

```
import tensorflow as tf
import numpy as np
import matplotlib.pyplot as plt

from tensorflow.examples.tutorials.mnist import input_data
mnist = input_data.read_data_sets('MNIST_data', one_hot=True)

x = tf.placeholder(tf.float32, shape=[None, 784])
y_ = tf.placeholder(tf.float32, shape=[None, 10])
```

We define now some tool functions to specify variable initialization.

```
def weight_variable(shape):
  initial = tf.truncated_normal(shape, stddev=0.1)
  return tf.Variable(initial)

def bias_variable(shape):
  initial = tf.constant(0.1, shape=shape)
  return tf.Variable(initial)
```

The following two function define convolution and (3-by-3) max pooling. The vector *strides* specifies how the filter or the sliding window move along each dimension. The vector *ksize* specifies the dimension of the sliding window. The padding option *'SAME'* automatically adds empty (zero valued) pixels to allow the convolution to be centered even in the boundary pixels.

```
def conv2d(x, W):
  return tf.nn.conv2d(x, W, strides=[1, 1, 1, 1], padding='SAME')

# max pooling over 3x3 blocks
```

```
def max_pool_3x3(x):
  return tf.nn.max_pool(x, ksize=[1, 3, 3, 1],
      strides=[1, 3, 3, 1], padding='SAME')
```

In order to define the model, we start by reshaping the input (where each sample is provided as 1-D vector) to its original size, i.e. each sample is represented by a matrix of 28x28 pixels. Then we define the first convolution layer which computes 12 features by using 5x5 filters. Finally we perform the `ReLU` activation and the first max pooling step.

```
# Input resize
x_image = tf.reshape(x, [-1,28,28,1])

# First convolution layer - 5x5 filters
INPUT_C1 = 1 # input channel
OUTPUT_C1 = 12 # output channel (features)
W_conv1 = weight_variable([5, 5, INPUT_C1, OUTPUT_C1])
b_conv1 = bias_variable([OUTPUT_C1])

#convolution step
h_conv1 = tf.nn.relu( conv2d(x_image, W_conv1) + b_conv1 )

#max pooling step
h_pool1 = max_pool_3x3(h_conv1)
```

The second convolution layer can be built up in an analogous way.

```
# Second convolution layer
INPUT_C2 = OUTPUT_C1
OUTPUT_C2 = 16 # output channel (features)
W_conv2 = weight_variable([5, 5, INPUT_C2, OUTPUT_C2])
b_conv2 = bias_variable([OUTPUT_C2])

#convolution step
h_conv2 = tf.nn.relu(conv2d(h_pool1, W_conv2) + b_conv2)

#max pooling step
h_pool2 = max_pool_3x3(h_conv2)
```

At this point the network returns 16 feature maps 4x4. These will be reshaped to 1–D vectors and given as input to the last fully connected linear layer. The linear layer is equipped with 1024 hidden units with $ReLU$ activation functions.

```
# Definition of the fully connected linear layer
FS = 4 # final size
W_fc1 = weight_variable([ FS * FS * OUTPUT_C2, 1024])
b_fc1 = bias_variable([1024])

# Reshape images
h_pool2_flat = tf.reshape(h_pool2, [-1, FS * FS * OUTPUT_C2])
```

```
# hidden layer
h_fc1 = tf.nn.relu(tf.matmul(h_pool2_flat, W_fc1) + b_fc1)

# Output layer
W_fc2 = weight_variable([1024, 10])
b_fc2 = bias_variable([10])

# Prediction
y_conv = tf.matmul(h_fc1, W_fc2) + b_fc2
```

In the following piece of code we report the optimization process of the defined CNN. As in the standard ANN case, it is organized in a `for` loop. This time, we chose the *Adam* gradient-based optimization by the function `AdamOptimizer`. Each epoch performs a training step over mini-batches extracted again by the dedicated function `next_batch()`, introduced in Section 4.3. This time the computations are run within *InteractiveSession*. The difference with the regular *Session* is that an *InteractiveSession* sets itself as the default session during building, allowing to run variables without needing to constantly refer to the session object. As a for instance, the method `eval()` will implicitly use that session to run operations.

```
cross_entropy = tf.reduce_mean(tf.nn.softmax_cross_entropy_with_logits(y_conv, y_))
train_step = tf.train.AdamOptimizer(1e-4).minimize(cross_entropy)
correct_prediction = tf.equal(tf.argmax(y_conv,1), tf.argmax(y_,1))
accuracy = tf.reduce_mean(tf.cast(correct_prediction, tf.float32))

sess = tf.InteractiveSession()
sess.run(tf.global_variables_initializer())

# Early stopping setup, to check on validation set
prec_err = 10**6 # just a very big value
val_count = 0
val_max_steps = 6

epochs = 100
batch_size = 1000
num_of_batches = 60000/batch_size

i=1
while i <= epochs and val_count < val_max_steps:

  print 'Epoch:',i,'(Early stopping criterion:',val_count,'/',val_max_steps,')'

  # training step
  for j in range(num_of_batches):
    batch = mnist.train.next_batch(batch_size)
    sess.run(train_step, feed_dict={x: batch[0], y_: batch[1]})

  # visualize accuracy each 10 epochs
  if i == 1 or i%10 == 0:
    train_accuracy = accuracy.eval(
          feed_dict={x: mnist.train.images, y_: mnist.train.labels})
    test_accuracy = accuracy.eval(
          feed_dict={x: mnist.test.images, y_: mnist.test.labels})
    print "\nAccuracy at epoch ", i, ":"
    print("train accuracy %g, test accuracy %g\n"%(train_accuracy, test_accuracy))
```

```
  # validation check
  curr_err = sess.run(cross_entropy,
        feed_dict={x: mnist.validation.images, y_: mnist.validation.labels})
  if curr_err >= prec_err*0.9999:
    val_count = val_count + 1
  else:
    val_count = 0
  prec_err = curr_err

  i+=1


print("\n\nResult:\nTest accuracy %g" % accuracy.eval(
      feed_dict={x: mnist.test.images, y_: mnist.test.labels}))
```

In this setting, the final classification accuracy on the test set should be close to the 99%. As in the previous Sections, in Figure 12 we show the learned filters of the first convolutional layer obtained by:

```
# Getting filters as an array
FILTERS = W_conv1.eval()

fig = plt.figure()
for i in range(np.shape(FILTERS)[3]):
  ax = fig.add_subplot(2, 6, i+1)
  ax.matshow(FILTERS[:,:,0,i], cmap='gray')
plt.show()
```

In the first line `eval()` computes the current value of `W_conv1`, saving it in the 4–D `numpy` array *FILTERS*, where the fourth dimension (acceded by the index 3 in `np.shape(FILTERS)[3]`) corresponds to the number of filters.

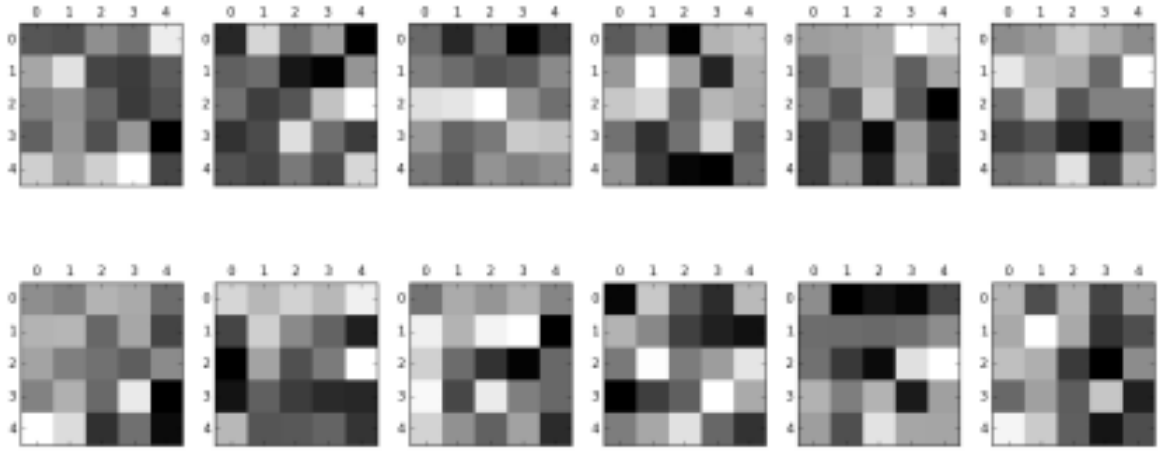


Figure 12: First Convolutional Layer filters of dimension 5×5 after the training with TensorFlow on the MNIST images with the described arcitecture.

6 A critical comparison

In this Section we would like to outline an overall picture across the presented environments. Even if in Table 1 we provide a scoring based on some features we thought mainly relevant for Machine Learning software development, this work would not like to bound this analysis to a poor evaluation. Instead, we hope to propose an useful guideline to help people trying to approach ANNs and Machine Learning in general, in order to orientate within the environments depending on personal background and requirements. More complete and statistically relevant comparisons can be found on the web[22], but we try to summarize so as to help and speed up single and global task developing.

We first give general description of each environment, then we try to compare pros and cons on specific requirements. At the end, we carry out an indicative numerical analysis on the computational performances on different tasks, which could be also a topic for comparison and discussion.

6.1 MATLAB

The programming language is intuitive and the software provides a complete package, allowing the user to define and train almost all kind of ANNs architecture without writing a single line of specific code. The code parallelization is automatic and the integration with CUDA® is straightforward too. The available built-in functions are very customizable and optimized, providing fast and extended setting up of experiments and an easy access to the variable of the network for in-depth analysis. However, enlarging and integrating MATLABtools require an advanced knowledge of the environment. This could drive the user to start rewriting its own code from the beginning, leading to a general decay of computational performances. These features make it perfect as a statistical and analysis toolbox, but maybe a bit slow as developmental environment. The GUI results sometimes heavy to be handled by the calculator, but, on the other hand, it is very user-friendly and provides the best graphical data visualization. The documentation is complete and well organized within the official site.

6.2 Torch

The programming language (Lua) can sometimes results a little bit tricky, but it supposed to be the faster among these languages. It provides all the needed CUDA® integrations and the CPU parallelization automatic. The module-based structure allows flexibility in the ANNs architecture and it is relatively easy to extend the provided packages. There are also other powerful packages[23], but in general they require to acquire some expertise to achieve a conscious handling. Torch could be easily used as a prototyping environment for specific and generic algorithms testing. The documentation is spread all over the `torch` GitHub repository and sometimes solve specific issues could not be immediate.

6.3 Tensor Flow

The employment of a programming language as dynamic as Python makes the code scripting light for the user. The CPU parallelization is automatic, and, exploiting the graph-structure of the computation is easy to take advantage of GPU computing. It provides a good data visualization and the possibility for beginners to access to ready to go packages, even if not treated in this document. The power of symbolic computation involves the user only in the forward step, whereas the backward step is entirely derived by the TensorFlow environment. This flexibility allows a very fast development for users from any level of expertise.

[22] Look for example at the webpage `http://hammerprinciple.com/therighttool`

[23] We skip the treatment of `optim`, which provides various Gradient Descent and Back-Propagation procedure

6.4 An overall picture on the comparison

As already said, in Table 1 we try to sum up a global comparison trying assigning a score from 1 to 5 on different perspectives. Here below, we explain the main motivation when necessary:

Programming Language All the basic language are quite intuitive.

GPU Integration MATLABis penalized since an extra toolbox is required.

CPU Parallelization All the environments exploit as more core as possible

Function Customizability MATLABscore is lower since integrate well-optimized functions with the provided ones is difficult

Symbolic Calculus Not expected in Lua

Network Structure Customizability Every kind of network is possible

Data Visualization The interactive MATLABmode outperforms the others

Installation Quite simple for all of them, but the MATLABinteractive GUI is an extra point

OS Compatibility Torch installation is not easy on Windows

Built-In Function Availability MATLABprovided simple-tools with an easy access

Language Performance MATLABinterface can sometimes appear heavy

Development Flexibility Again, MATLABis penalized because it forces medium users to become very specialized with the language to integrate the provided tools or to write proper code, which in general can slow down the software development

	MATLAB	Torch	TensorFlow
Programming Language	4	3	4
GPU Integration	3	5	5
CPU Parallelization	5	5	5
Function Customizability	2	4	5
Symbolic Calculus	3	1	5
Network Structure Customizability	5	5	5
Data Visualization	5	2	3
Installation	5	4	4
OS Compatibility	5	4	5
Built-In Function Availability	5	4	4
Language Performance	3	5	4
Development Flexibility	2	4	5
License	EULA	BSD (Open source)	Apache 2.0 (Open source)

Table 1: Environments individual scoring

6.5 Computational issues

In Table 2 we compare running times for different tasks, analyzing the advantages and differences of CPU/GPU computing. Results are averaged on 5 trials, carried out in the same machine with an Intel(R) Xeon(R) CPU E5-2650 v2 @ 2.60GHz with 32 cores, 66 GB of RAM, and a Geforce GTX 960 with 4GB of memory. The OS is Debian GNU/Linux 8 (jessie). We test a standard Gradient Descent procedure varying the network architecture, the batches size (among *Stochastic Gradient Descent* (SGD), 1000 samples batch and Full Batch) and the hardware (indicated in HW column). The CNN architecture is the same of the one proposed in Figure 8. Performances are obtained trying to use optimization procedures as similar as possible. In practice, it is very difficult to reply the specific optimization techniques exploited in MATLABbuilt-in toolboxes. We skip the SGD case for the second architecture (eighth row) in Torch because of the huge computational time obtained for the first architecture. We miss the SGD case for the ANNs architecture in the MATLABcase since the training function *'trains'* it is not supported for GPU computing (rows fourth and tenth). As a matter of fact, this could be an uncommon case of study, but we report the results for best completeness. We skip the CNN Full Batch trials on GPU because of the too large memory requirement[24] .

[24]For an exhaustive comparison on computational performance on several tasks (including the comparison with other existent packages) the user can refer to `https://en.wikipedia.org/wiki/Comparison_of_deep_learning_software`

Architecture	Batches size			Env.	HW
	SGD	1000	Full Batch		
2-Layers ANN 1000 HUs	52.46	30.40	28.38	Matlab	CPU
	3481.00	46.55	24.43	Torch	
	914.56	13.82	12.00	TF	
	–	3.75	3.73		GPU
	378.19	1.93	1.46		
	911.32	5.40	4.92		
4-Layers ANN 300-300-300 HUs	44.66	28.33	27.07		CPU
	–	52.99	20.00		
	893.27	10.95	8.83		
	–	2.74	3.44		GPU
	517.87	1.51	1.08		
	1024.29	4.73	4.27		
CNN	7794.33	54.21	–		GPU
	647.75	100.06	–		
	1850.30	20.22	–		

Table 2: Averaged time (in seconds) on 5 running of 10 epochs training with different architectures on the MNIST data within the presented environments. All the architectures are built up using the *ReLU* as activation, the *softmax* as output function and the *Cross-Entropy* penalty.

References

[1] Christopher M. Bishop. *Pattern Recognition and Machine Learning*. Springer, 2006.

[2] David E Rumelhart, Geoffrey E Hinton, and Ronald J Williams. Learning representations by back-propagating errors. *Cognitive modeling*, 5(3):1, 1988.

[3] F. Giannini, V. Laveglia, A. Rossi, D. Zanca, and A. Zugarini. *NeuralNetworksForBeginners*. `https://github.com/AILabUSiena/NeuralNetworksForBeginners`, 2016.

[4] The Mathworks, Inc., Natick, Massachusetts. *MATLAB version 8.5.0.197613 (R2015a)*, 2015.

[5] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, et al. Tensorflow: Large-scale machine learning on heterogeneous distributed systems. *arXiv preprint arXiv:1603.04467*, 2016.

[6] Y. Lecun, L. Bottou, Y. Bengio, and P. Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, Nov 1998.

[7] Kunihiko Fukushima. Neocognitron: A hierarchical neural network capable of visual pattern recognition. *Neural networks*, 1(2):119–130, 1988.

[8] Thomas Serre, Lior Wolf, Stanley Bileschi, Maximilian Riesenhuber, and Tomaso Poggio. Robust object recognition with cortex-like mechanisms. *IEEE transactions on pattern analysis and machine intelligence*, 29(3), 2007.