Designing a convolutional neural network for hand-written digits classification

Michele Marolla

This is the technical project for *Machine Learning and Applications* exam, at University Federico II, Naples.

A generic convolutional neural network is designed and implemented from scratch as a stand-alone library. Then, it is trained in order to recognize raw images of hand-written digits, and its performances are evaluated. In particular, the *MNIST* dataset has been used (for further information, see here: http://yann.lecun.com/exdb/mnist/).

For this project, *MATLAB* has been chosen due to its strong support on matrix algebra. This document has been written using *MATLAB Live Script.*

Designing and implementing a multi-layer neural network

Network is a handle class that implements a multi-layer feed-forward neural network. Every *Network* has two public properties: a *name* string, that let the user define the name of the network, and a cell array of *Layer* objects.

Layer is a value class that implements a single layer of nodes. It contains a matrix of weights and biases, values used for *resilient back-propagation*, and names and pointers to activation and output functions. It is possible to arbitrarly assign any activation and output function to each layer: by default, sigmoid is used for internal layers and identity for output layers.

For further information on the single functions, their parameters and default values, see the .m code files: they are extensively commented.

Forward propagation

In order to create a new *Layer*, you need to define a weight matrix (dimensions equal to number of nodes x numbers of inputs), biases column vector (dimensions equal to number of nodes), the activation function and output function (both as strings). The identity function (*"identity"*), the Rectified Linear Unit (*"ReLU"*) and the sigmoid (*"sigmoid"*) have already been implemented. However, you can define whatever function you want inside a .m file and pass their name. Sigmoid and identity .m function file has been created as an example.

```
%% Creating a layer with 3 nodes and 5 inputs, using sigmoid as activation function
%% and identity as output function
W = randn(3,5);
B = randn(3,1);
MyLayer = Layer(W,B,"sigmoid","identity");
```

Layer's function *compute* is used to compute the output of that single layer, given the input. The input must be in form a vector, with dimension coherent with the weight matrix (it can be indifferently a row or column vector). The output is returned as a column vector, with dimension equal to the number of nodes.

```
%% Example: compute
Input = randn(1,5);
LayerOutput = MyLayer.compute(Input);
```

In order to create a *Network* you need to specify:

- Input dimension (integer)
- Layers number and dimensions as a vector containing the number of nodes for each layer (example: *[3 2]* defines two layers, the first with 3 nodes and the second with 2 nodes).

Other optional parameters are:

- Output function of each layer, as a vector of string. Identity is used by default.
- Activation function of each layer, as a vector of string. By default, sigmoid is used for internal layers and identity for the last layer.
- Error function used for learning: cross entropy with softmax (default) and sum of squares have already been implemented, but the user can define an arbitrary function.
- The derivative of the error function; this must be specified if the user has defined an own error function.
- A cell array of dimensions equal to the number of layers containing matrices that specify which connection are enabled. Each matrix must have the same dimensions of layers' weight matrices: the element *(i,j)* must have value 1 if there is a connection between node *i* and node *j*, 0 otherwise. By default, the network is full connected (i.e., the matrix has all the elements equal to 1).

```
% Creating a full-connected network with 3 layers, with 3, 4 and 5 nodes
% respectively, and 10 inputs.

MyNetwork = Network(10, [3,4,5]);
MyNetwork.name = 'MyNetwork';
```

Network's *forward* function computes the forward propagation inside the network, given an input vector of suitable dimension (equals to the number of columns of the weight matrix of the first layer). Two values are returned: a row vector containing the outputs of the last layer, and a cell array containing the outputs (as row vectors) of each internal layer.

```
% Example: forward

Input = randn(1,10);
[output, LayersOutput] = MyNetwork.forward(Input);
```

The *reset* function resets weights and biases with random values for each layer.

```
% Example: reset
MyNetwork.reset();
```

The *setConnectionMap* function takes a cell array that specify which connection to enable / disable in the network. Again, value 1 enable the connection and 0 disable it.

```
% Example: setConnectionMap. Remove the connection of the first node of first layer
% with the first 5 inputs
for i = 1:length(MyNetwork.layers)
    FC{i} = ones(size(MyNetwork.layers{i}.W));
end
FC{1}(1,1:5) = 0;
MyNetwork.setConnectionMap(FC);
```

Back propagation

The learning phase is activated by using the **learn** function. Besides the training and validation set, it also accepts a number of optional parameters, so that the user can specify:

- the number of epochs (default: 1000)
- the learning rate η (scalar) for the gradient descent method, or the vector $[\eta^- \ \eta^+]$ for the resilient back-propagation (default: gradient descent with $\eta = 0.0005$)
- in case of mini-batch approach, the number of elements for each subdivision; if equal to -1, the batch approach is used (default: -1)
- weight-decay coefficient (default: 0)
- moment coefficient (default: 0)
- an early stopping criterion (default: empty). At the current state, two different criteria has been implemented: *"firstMin"*, for which the learning is ended when the first minimum is encountered; *"stopEpochs"*, for which the learning is ended if the validation error does not improve after a specified number of consecutive epochs.

The learn functions returns three outputs: an array of error values on the training set for each epoch, an array of errors on the validation set, and the best network corresponding to the minimum error on the validation set.

In the next two paragraphs, you can find some examples where the learn function is used.

To implement the learning phase, two private function has been implemented too: one for computing the δ values for each node, and one for computing the derivatives of the error function through back-propagation. For further information, see the comments in the *Network.m* file.

The user can specify an arbitrary error function through *setErrorFunction*, that takes three input parameters:

- the error function name, that could be "*crossEntropy"* or *"sumOfSquares"* (these are already implemented), or the name of a function written by the user
- the name of the function that computes the derivative of the error (mandatory only for user-defined functions)
- a flag that enable or disable a softmax post-processing (mandatory only for user-defined functions)

For example, in order to specify an error function that needs softmax, you can write:

```
MyNetwork.setErrorFunction("myError", "myErrorDerivative", 1);
```

A test on a simple dataset: IRIS

We want now to test the network on a simple dataset: IRIS contains information on three different species of flowers. It contains 150 entries, each of them with 4 different features.

```
DS = readmatrix('iris.data', 'FileType', 'text');
X = DS(:,1:4);
T = (getTargetsFromLabels(DS(:,5)))';
```

Let's build a network called *'iris_net'*, with 4 inputs, 10 internal nodes and 3 outputs:

```
net = Network(4, [10 3]);
net.name = "iris_net";
```

A random shuffle of the dataset is executed; then, 100 elements are used for the training set, and 50 for the validation set.

```
% Shuffle
perm = randperm(length(X));
X = X(perm,:);
T = T(perm,:);

% Training set
X_t = X(1:100,:);
T_t = T(1:100,:);

% Validation set
X_v = X(101:150,:);
T_v = T(101:150,:);
```

The network will now learn using a batch approach, learning rate $\eta = 0.0005$, for 200 epochs. Training and validation error are plotted.

```
[t_err, v_err, best_net] = net.learn(X_t, T_t, X_v, T_v, 200, 0.0005);
```

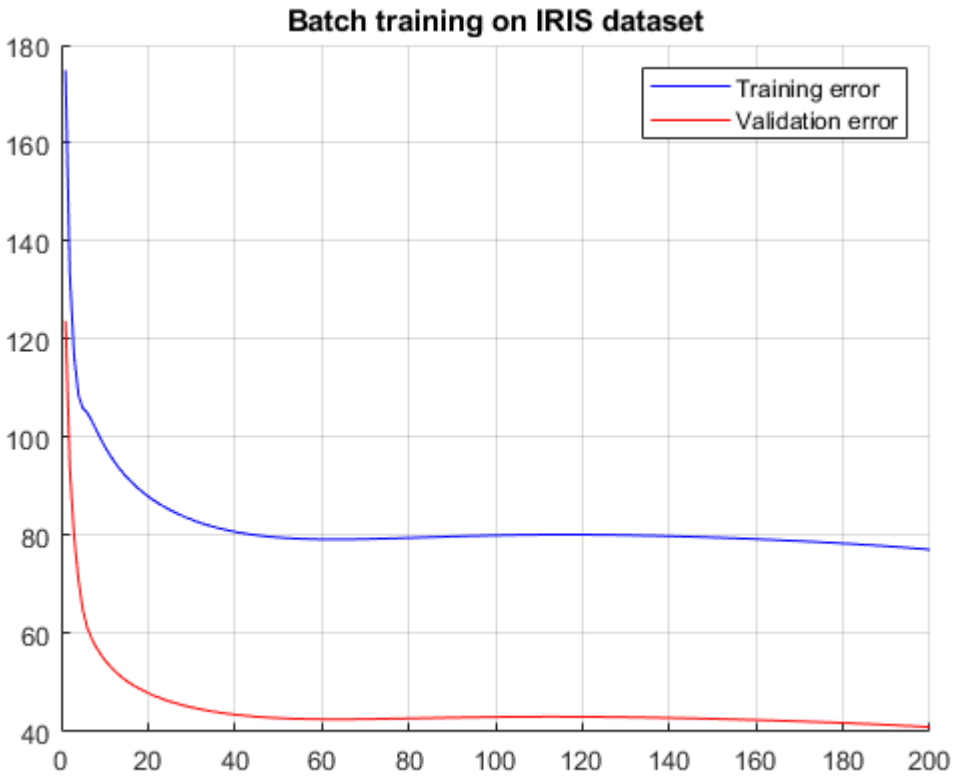
```
init err: 242.6971
epoch: 1, train_err: 174.8949, val_err: 123.7709
epoch: 2, train_err: 133.1531, val_err: 93.5188
epoch: 3, train_err: 116.4869, val_err: 79.7034
epoch: 4, train_err: 108.5775, val_err: 70.8081
epoch: 5, train_err: 105.9817, val_err: 64.8145
epoch: 6, train_err: 105.0416, val_err: 61.2735
epoch: 7, train_err: 103.5415, val_err: 59.0438
epoch: 8, train_err: 101.7178, val_err: 57.359
epoch: 9, train_err: 99.9225, val_err: 55.9496
epoch: 10, train_err: 98.2647, val_err: 54.7283
epoch: 11, train_err: 96.7602, val_err: 53.6569
epoch: 12, train_err: 95.3991, val_err: 52.7111
epoch: 13, train_err: 94.1658, val_err: 51.8717
epoch: 14, train_err: 93.0451, val_err: 51.1226
epoch: 15, train_err: 92.0229, val_err: 50.4503
```

```
epoch: 16, train_err: 91.0871, val_err: 49.8435
epoch: 17, train_err: 90.2271, val_err: 49.2928
epoch: 18, train_err: 89.4343, val_err: 48.7905
epoch: 19, train_err: 88.701, val_err: 48.3303
epoch: 20, train_err: 88.0209, val_err: 47.9067
epoch: 21, train_err: 87.3885, val_err: 47.5154
epoch: 22, train_err: 86.7993, val_err: 47.1529
epoch: 23, train_err: 86.2494, val_err: 46.8161
epoch: 24, train_err: 85.7353, val_err: 46.5024
epoch: 25, train_err: 85.2541, val_err: 46.2097
epoch: 26, train_err: 84.8034, val_err: 45.9362
epoch: 27, train_err: 84.3809, val_err: 45.6803
epoch: 28, train_err: 83.9847, val_err: 45.4407
epoch: 29, train_err: 83.613, val_err: 45.2162
epoch: 30, train_err: 83.2644, val_err: 45.0058
epoch: 31, train_err: 82.9374, val_err: 44.8085
epoch: 32, train_err: 82.6308, val_err: 44.6236
epoch: 33, train_err: 82.3435, val_err: 44.4504
epoch: 34, train_err: 82.0745, val_err: 44.2881
epoch: 35, train_err: 81.8227, val_err: 44.1362
epoch: 36, train_err: 81.5873, val_err: 43.9942
epoch: 37, train_err: 81.3676, val_err: 43.8615
epoch: 38, train_err: 81.1626, val_err: 43.7377
epoch: 39, train_err: 80.9718, val_err: 43.6223
epoch: 40, train_err: 80.7943, val_err: 43.5149
epoch: 41, train_err: 80.6296, val_err: 43.4152
epoch: 42, train_err: 80.4771, val_err: 43.3228
epoch: 43, train_err: 80.3362, val_err: 43.2372
epoch: 44, train_err: 80.2062, val_err: 43.1583
epoch: 45, train_err: 80.0868, val_err: 43.0857
epoch: 46, train_err: 79.9773, val_err: 43.0191
epoch: 47, train_err: 79.8773, val_err: 42.9582
epoch: 48, train_err: 79.7863, val_err: 42.9027
epoch: 49, train_err: 79.7038, val_err: 42.8524
epoch: 50, train_err: 79.6295, val_err: 42.8069
epoch: 51, train_err: 79.5629, val_err: 42.7662
epoch: 52, train_err: 79.5035, val_err: 42.7298
epoch: 53, train_err: 79.4511, val_err: 42.6976
epoch: 54, train_err: 79.4052, val_err: 42.6694
epoch: 55, train_err: 79.3655, val_err: 42.6449
epoch: 56, train_err: 79.3315, val_err: 42.624
epoch: 57, train_err: 79.3031, val_err: 42.6064
epoch: 58, train_err: 79.2798, val_err: 42.5919
epoch: 59, train_err: 79.2613, val_err: 42.5804
epoch: 60, train_err: 79.2474, val_err: 42.5716
epoch: 61, train_err: 79.2377, val_err: 42.5654
epoch: 62, train_err: 79.232, val_err: 42.5616
epoch: 63, train_err: 79.23, val_err: 42.56
epoch: 64, train_err: 79.2314, val_err: 42.5605
epoch: 65, train_err: 79.2359, val_err: 42.563
epoch: 66, train_err: 79.2435, val_err: 42.5672
epoch: 67, train_err: 79.2537, val_err: 42.573
epoch: 68, train_err: 79.2664, val_err: 42.5804
epoch: 69, train_err: 79.2814, val_err: 42.5891
epoch: 70, train_err: 79.2985, val_err: 42.599
epoch: 71, train_err: 79.3174, val_err: 42.6101
epoch: 72, train_err: 79.338, val_err: 42.6221
epoch: 73, train_err: 79.3601, val_err: 42.6351
epoch: 74, train_err: 79.3836, val_err: 42.6488
epoch: 75, train_err: 79.4082, val_err: 42.6631
epoch: 76, train_err: 79.4339, val_err: 42.6781
epoch: 77, train_err: 79.4604, val_err: 42.6935
epoch: 78, train_err: 79.4876, val_err: 42.7093
epoch: 79, train_err: 79.5154, val_err: 42.7254
```

```
epoch: 80, train_err: 79.5436, val_err: 42.7418
epoch: 81, train_err: 79.5722, val_err: 42.7583
epoch: 82, train_err: 79.601, val_err: 42.7749
epoch: 83, train_err: 79.6299, val_err: 42.7915
epoch: 84, train_err: 79.6588, val_err: 42.808
epoch: 85, train_err: 79.6876, val_err: 42.8244
epoch: 86, train_err: 79.7161, val_err: 42.8407
epoch: 87, train_err: 79.7444, val_err: 42.8567
epoch: 88, train_err: 79.7724, val_err: 42.8725
epoch: 89, train_err: 79.7998, val_err: 42.8879
epoch: 90, train_err: 79.8268, val_err: 42.903
epoch: 91, train_err: 79.8532, val_err: 42.9177
epoch: 92, train_err: 79.8789, val_err: 42.9319
epoch: 93, train_err: 79.9039, val_err: 42.9457
epoch: 94, train_err: 79.9281, val_err: 42.9589
epoch: 95, train_err: 79.9515, val_err: 42.9716
epoch: 96, train_err: 79.9741, val_err: 42.9838
epoch: 97, train_err: 79.9957, val_err: 42.9953
epoch: 98, train_err: 80.0164, val_err: 43.0063
epoch: 99, train_err: 80.0361, val_err: 43.0166
epoch: 100, train_err: 80.0548, val_err: 43.0263
epoch: 101, train_err: 80.0724, val_err: 43.0353
epoch: 102, train_err: 80.089, val_err: 43.0436
epoch: 103, train_err: 80.1045, val_err: 43.0513
epoch: 104, train_err: 80.1189, val_err: 43.0582
epoch: 105, train_err: 80.1322, val_err: 43.0644
epoch: 106, train_err: 80.1443, val_err: 43.0699
epoch: 107, train_err: 80.1552, val_err: 43.0747
epoch: 108, train_err: 80.1651, val_err: 43.0788
epoch: 109, train_err: 80.1737, val_err: 43.0821
epoch: 110, train_err: 80.1812, val_err: 43.0847
epoch: 111, train_err: 80.1875, val_err: 43.0866
epoch: 112, train_err: 80.1926, val_err: 43.0877
epoch: 113, train_err: 80.1966, val_err: 43.0881
epoch: 114, train_err: 80.1994, val_err: 43.0878
epoch: 115, train_err: 80.201, val_err: 43.0868
epoch: 116, train_err: 80.2015, val_err: 43.085
epoch: 117, train_err: 80.2008, val_err: 43.0825
epoch: 118, train_err: 80.199, val_err: 43.0793
epoch: 119, train_err: 80.1961, val_err: 43.0754
epoch: 120, train_err: 80.192, val_err: 43.0708
epoch: 121, train_err: 80.1868, val_err: 43.0656
epoch: 122, train_err: 80.1806, val_err: 43.0596
epoch: 123, train_err: 80.1733, val_err: 43.053
epoch: 124, train_err: 80.1649, val_err: 43.0457
epoch: 125, train_err: 80.1554, val_err: 43.0378
epoch: 126, train_err: 80.145, val_err: 43.0292
epoch: 127, train_err: 80.1335, val_err: 43.0199
epoch: 128, train_err: 80.121, val_err: 43.0101
epoch: 129, train_err: 80.1075, val_err: 42.9996
epoch: 130, train_err: 80.0931, val_err: 42.9886
epoch: 131, train_err: 80.0777, val_err: 42.9769
epoch: 132, train_err: 80.0614, val_err: 42.9647
epoch: 133, train_err: 80.0442, val_err: 42.9518
epoch: 134, train_err: 80.026, val_err: 42.9384
epoch: 135, train_err: 80.007, val_err: 42.9245
epoch: 136, train_err: 79.9872, val_err: 42.91
epoch: 137, train_err: 79.9664, val_err: 42.895
epoch: 138, train_err: 79.9449, val_err: 42.8795
epoch: 139, train_err: 79.9225, val_err: 42.8634
epoch: 140, train_err: 79.8994, val_err: 42.8468
epoch: 141, train_err: 79.8754, val_err: 42.8298
epoch: 142, train_err: 79.8507, val_err: 42.8122
epoch: 143, train_err: 79.8253, val_err: 42.7942
```

```
epoch: 144, train_err: 79.7991, val_err: 42.7757
epoch: 145, train_err: 79.7722, val_err: 42.7567
epoch: 146, train_err: 79.7445, val_err: 42.7373
epoch: 147, train_err: 79.7162, val_err: 42.7174
epoch: 148, train_err: 79.6871, val_err: 42.697
epoch: 149, train_err: 79.6574, val_err: 42.6763
epoch: 150, train_err: 79.6271, val_err: 42.6551
epoch: 151, train_err: 79.596, val_err: 42.6334
epoch: 152, train_err: 79.5643, val_err: 42.6114
epoch: 153, train_err: 79.532, val_err: 42.5889
epoch: 154, train_err: 79.499, val_err: 42.566
epoch: 155, train_err: 79.4654, val_err: 42.5427
epoch: 156, train_err: 79.4312, val_err: 42.519
epoch: 157, train_err: 79.3963, val_err: 42.4949
epoch: 158, train_err: 79.3609, val_err: 42.4703
epoch: 159, train_err: 79.3248, val_err: 42.4454
epoch: 160, train_err: 79.288, val_err: 42.4201
epoch: 161, train_err: 79.2507, val_err: 42.3943
epoch: 162, train_err: 79.2127, val_err: 42.3681
epoch: 163, train_err: 79.1741, val_err: 42.3416
epoch: 164, train_err: 79.1349, val_err: 42.3146
epoch: 165, train_err: 79.095, val_err: 42.2872
epoch: 166, train_err: 79.0545, val_err: 42.2593
epoch: 167, train_err: 79.0134, val_err: 42.2311
epoch: 168, train_err: 78.9716, val_err: 42.2024
epoch: 169, train_err: 78.9291, val_err: 42.1732
epoch: 170, train_err: 78.8859, val_err: 42.1436
epoch: 171, train_err: 78.842, val_err: 42.1136
epoch: 172, train_err: 78.7974, val_err: 42.0831
epoch: 173, train_err: 78.7521, val_err: 42.0521
epoch: 174, train_err: 78.706, val_err: 42.0206
epoch: 175, train_err: 78.6592, val_err: 41.9887
epoch: 176, train_err: 78.6115, val_err: 41.9562
epoch: 177, train_err: 78.5631, val_err: 41.9232
epoch: 178, train_err: 78.5138, val_err: 41.8897
epoch: 179, train_err: 78.4636, val_err: 41.8556
epoch: 180, train_err: 78.4126, val_err: 41.8209
epoch: 181, train_err: 78.3606, val_err: 41.7857
epoch: 182, train_err: 78.3077, val_err: 41.7498
epoch: 183, train_err: 78.2537, val_err: 41.7134
epoch: 184, train_err: 78.1988, val_err: 41.6763
epoch: 185, train_err: 78.1427, val_err: 41.6385
epoch: 186, train_err: 78.0856, val_err: 41.6
epoch: 187, train_err: 78.0273, val_err: 41.5609
epoch: 188, train_err: 77.9679, val_err: 41.521
epoch: 189, train_err: 77.9072, val_err: 41.4804
epoch: 190, train_err: 77.8452, val_err: 41.4389
epoch: 191, train_err: 77.7819, val_err: 41.3967
epoch: 192, train_err: 77.7172, val_err: 41.3536
epoch: 193, train_err: 77.6511, val_err: 41.3097
epoch: 194, train_err: 77.5835, val_err: 41.2649
epoch: 195, train_err: 77.5144, val_err: 41.2192
epoch: 196, train_err: 77.4437, val_err: 41.1725
epoch: 197, train_err: 77.3713, val_err: 41.1248
epoch: 198, train_err: 77.2972, val_err: 41.0762
epoch: 199, train_err: 77.2213, val_err: 41.0264
epoch: 200, train_err: 77.1436, val_err: 40.9756
```

```
figure(), grid, hold on
plot(t_err, 'b'), plot(v_err, 'r')
legend('Training error', 'Validation error')
title('Batch training on IRIS dataset')
```



Let's repeat the test using a mini-batch approach, with 32 elements in each batch:

```
[t_err, v_err, best_net] = net.learn(X_t, T_t, X_v, T_v, 200, 0.0005, 32);
```

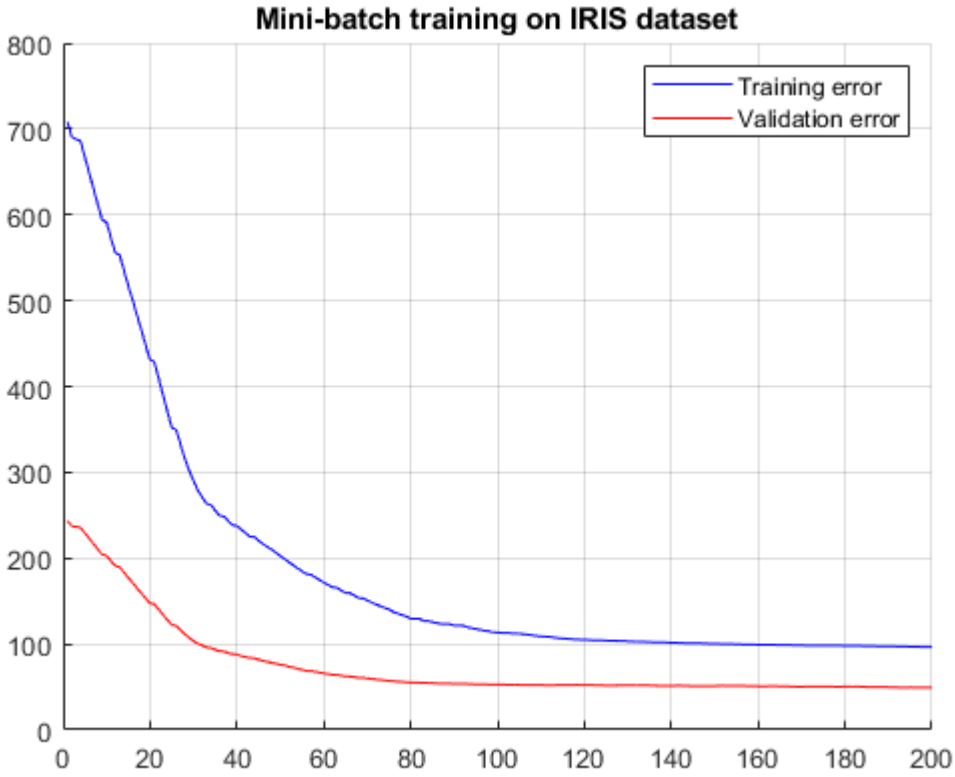
```
init err: 724.2842
epoch: 1, train_err: 708.2575, val_err: 243.3775
epoch: 2, train_err: 690.5014, val_err: 237.3201
epoch: 3, train_err: 688.17, val_err: 236.4556
epoch: 4, train_err: 685.8303, val_err: 235.5889
epoch: 5, train_err: 667.3556, val_err: 229.327
epoch: 6, train_err: 648.4922, val_err: 222.9415
epoch: 7, train_err: 630.6863, val_err: 216.7664
epoch: 8, train_err: 611.6122, val_err: 210.2785
epoch: 9, train_err: 593.9544, val_err: 204.1068
epoch: 10, train_err: 591.598, val_err: 203.2289
epoch: 11, train_err: 572.8697, val_err: 196.7881
epoch: 12, train_err: 555.7702, val_err: 190.7365
epoch: 13, train_err: 553.5341, val_err: 189.8894
epoch: 14, train_err: 535.3218, val_err: 183.5571
epoch: 15, train_err: 517.2458, val_err: 177.2566
epoch: 16, train_err: 500.8577, val_err: 171.3982
epoch: 17, train_err: 482.8303, val_err: 165.1466
epoch: 18, train_err: 466.4515, val_err: 159.3621
epoch: 19, train_err: 447.9331, val_err: 153.1013
epoch: 20, train_err: 431.0932, val_err: 147.3541
epoch: 21, train_err: 429.0463, val_err: 146.5951
epoch: 22, train_err: 409.5713, val_err: 140.3302
epoch: 23, train_err: 389.601, val_err: 134.1088
epoch: 24, train_err: 371.3035, val_err: 127.9911
epoch: 25, train_err: 351.6936, val_err: 122.2575
epoch: 26, train_err: 349.7979, val_err: 121.6139
```

```
epoch: 27, train_err: 333.0434, val_err: 116.2605
epoch: 28, train_err: 316.098, val_err: 111.6172
epoch: 29, train_err: 302.3726, val_err: 107.3258
epoch: 30, train_err: 290.312, val_err: 103.5625
epoch: 31, train_err: 279.196, val_err: 100.7258
epoch: 32, train_err: 271.1687, val_err: 98.5258
epoch: 33, train_err: 263.2038, val_err: 95.902
epoch: 34, train_err: 262.4182, val_err: 95.6093
epoch: 35, train_err: 255.5149, val_err: 93.2763
epoch: 36, train_err: 249.4158, val_err: 91.7075
epoch: 37, train_err: 248.7842, val_err: 91.4529
epoch: 38, train_err: 243.2495, val_err: 89.4513
epoch: 39, train_err: 238.3383, val_err: 88.1332
epoch: 40, train_err: 237.7933, val_err: 87.8995
epoch: 41, train_err: 233.0401, val_err: 86.0854
epoch: 42, train_err: 229.2909, val_err: 84.9128
epoch: 43, train_err: 225.1734, val_err: 83.747
epoch: 44, train_err: 224.6875, val_err: 83.5284
epoch: 45, train_err: 220.4773, val_err: 81.8516
epoch: 46, train_err: 216.4667, val_err: 80.2622
epoch: 47, train_err: 213.1874, val_err: 79.2107
epoch: 48, train_err: 209.9413, val_err: 78.1675
epoch: 49, train_err: 206.2797, val_err: 77.1175
epoch: 50, train_err: 202.6391, val_err: 76.0745
epoch: 51, train_err: 199.0068, val_err: 75.0372
epoch: 52, train_err: 195.2554, val_err: 73.5544
epoch: 53, train_err: 191.6797, val_err: 72.558
epoch: 54, train_err: 188.1046, val_err: 71.1838
epoch: 55, train_err: 184.6996, val_err: 69.9109
epoch: 56, train_err: 181.4547, val_err: 68.7328
epoch: 57, train_err: 181.1126, val_err: 68.593
epoch: 58, train_err: 177.6599, val_err: 67.698
epoch: 59, train_err: 174.7461, val_err: 66.8705
epoch: 60, train_err: 171.687, val_err: 65.8081
epoch: 61, train_err: 168.8439, val_err: 65.0319
epoch: 62, train_err: 165.957, val_err: 64.068
epoch: 63, train_err: 165.665, val_err: 63.9569
epoch: 64, train_err: 162.2998, val_err: 63.1665
epoch: 65, train_err: 159.6043, val_err: 62.3057
epoch: 66, train_err: 159.3369, val_err: 62.2073
epoch: 67, train_err: 156.7942, val_err: 61.4268
epoch: 68, train_err: 153.5369, val_err: 60.7175
epoch: 69, train_err: 153.2933, val_err: 60.6304
epoch: 70, train_err: 150.7334, val_err: 60.0308
epoch: 71, train_err: 148.4201, val_err: 59.3537
epoch: 72, train_err: 146.2344, val_err: 58.7374
epoch: 73, train_err: 144.1648, val_err: 58.1748
epoch: 74, train_err: 142.2016, val_err: 57.6597
epoch: 75, train_err: 140.3364, val_err: 57.1867
epoch: 76, train_err: 137.4409, val_err: 56.6668
epoch: 77, train_err: 135.75, val_err: 56.2533
epoch: 78, train_err: 133.6682, val_err: 55.8604
epoch: 79, train_err: 132.1293, val_err: 55.4957
epoch: 80, train_err: 129.5895, val_err: 55.1239
epoch: 81, train_err: 129.4621, val_err: 55.0816
epoch: 82, train_err: 129.3354, val_err: 55.0396
epoch: 83, train_err: 126.9652, val_err: 54.733
epoch: 84, train_err: 126.8498, val_err: 54.6931
epoch: 85, train_err: 125.6116, val_err: 54.4085
epoch: 86, train_err: 124.4515, val_err: 54.1518
epoch: 87, train_err: 123.3622, val_err: 53.9195
epoch: 88, train_err: 123.2692, val_err: 53.8874
epoch: 89, train_err: 123.1765, val_err: 53.8555
epoch: 90, train_err: 121.6902, val_err: 53.6609
```

```
epoch: 91, train_err: 121.6048, val_err: 53.63
epoch: 92, train_err: 121.5198, val_err: 53.5992
epoch: 93, train_err: 119.6462, val_err: 53.4555
epoch: 94, train_err: 118.4246, val_err: 53.3403
epoch: 95, train_err: 117.6157, val_err: 53.1552
epoch: 96, train_err: 116.5268, val_err: 53.0735
epoch: 97, train_err: 115.5308, val_err: 53.0156
epoch: 98, train_err: 114.8596, val_err: 52.8464
epoch: 99, train_err: 113.5759, val_err: 52.871
epoch: 100, train_err: 113.5305, val_err: 52.8429
epoch: 101, train_err: 112.9672, val_err: 52.6883
epoch: 102, train_err: 112.9258, val_err: 52.6618
epoch: 103, train_err: 112.4017, val_err: 52.5223
epoch: 104, train_err: 112.3638, val_err: 52.4971
epoch: 105, train_err: 111.8741, val_err: 52.3705
epoch: 106, train_err: 111.4135, val_err: 52.2541
epoch: 107, train_err: 110.979, val_err: 52.1464
epoch: 108, train_err: 110.2746, val_err: 52.1337
epoch: 109, train_err: 109.3185, val_err: 52.2241
epoch: 110, train_err: 108.9705, val_err: 52.1065
epoch: 111, train_err: 108.6415, val_err: 51.997
epoch: 112, train_err: 108.3296, val_err: 51.8948
epoch: 113, train_err: 107.5083, val_err: 52.0042
epoch: 114, train_err: 106.7996, val_err: 52.1359
epoch: 115, train_err: 106.4336, val_err: 52.1634
epoch: 116, train_err: 106.1926, val_err: 52.0274
epoch: 117, train_err: 105.6244, val_err: 52.1816
epoch: 118, train_err: 105.1372, val_err: 52.3465
epoch: 119, train_err: 105.1243, val_err: 52.3122
epoch: 120, train_err: 104.9171, val_err: 52.15
epoch: 121, train_err: 104.6802, val_err: 52.1769
epoch: 122, train_err: 104.482, val_err: 52.0146
epoch: 123, train_err: 104.296, val_err: 51.8636
epoch: 124, train_err: 104.2869, val_err: 51.8325
epoch: 125, train_err: 104.1118, val_err: 51.693
epoch: 126, train_err: 103.8639, val_err: 51.7155
epoch: 127, train_err: 103.8554, val_err: 51.6843
epoch: 128, train_err: 103.4425, val_err: 51.8518
epoch: 129, train_err: 103.4335, val_err: 51.8177
epoch: 130, train_err: 103.0791, val_err: 51.9901
epoch: 131, train_err: 102.9125, val_err: 51.8122
epoch: 132, train_err: 102.5961, val_err: 51.9847
epoch: 133, train_err: 102.5863, val_err: 51.9462
epoch: 134, train_err: 102.4603, val_err: 51.9574
epoch: 135, train_err: 102.2942, val_err: 51.7643
epoch: 136, train_err: 102.1406, val_err: 51.5848
epoch: 137, train_err: 101.9975, val_err: 51.4178
epoch: 138, train_err: 101.9915, val_err: 51.3838
epoch: 139, train_err: 101.8579, val_err: 51.2301
epoch: 140, train_err: 101.5337, val_err: 51.3958
epoch: 141, train_err: 101.4053, val_err: 51.4037
epoch: 142, train_err: 101.1527, val_err: 51.5754
epoch: 143, train_err: 101.0103, val_err: 51.3822
epoch: 144, train_err: 100.8788, val_err: 51.2027
epoch: 145, train_err: 100.7568, val_err: 51.0356
epoch: 146, train_err: 100.7524, val_err: 51.0013
epoch: 147, train_err: 100.7482, val_err: 50.9673
epoch: 148, train_err: 100.4766, val_err: 51.1328
epoch: 149, train_err: 100.3779, val_err: 51.1331
epoch: 150, train_err: 100.291, val_err: 51.1342
epoch: 151, train_err: 100.0941, val_err: 51.3061
epoch: 152, train_err: 99.931, val_err: 51.4758
epoch: 153, train_err: 99.9188, val_err: 51.4303
epoch: 154, train_err: 99.9071, val_err: 51.3853
```

```
epoch: 155, train_err: 99.7554, val_err: 51.1635
epoch: 156, train_err: 99.5972, val_err: 51.3316
epoch: 157, train_err: 99.4482, val_err: 51.1066
epoch: 158, train_err: 99.4008, val_err: 51.0933
epoch: 159, train_err: 99.3595, val_err: 51.0809
epoch: 160, train_err: 99.206, val_err: 50.8593
epoch: 161, train_err: 99.161, val_err: 50.8523
epoch: 162, train_err: 99.1507, val_err: 50.8094
epoch: 163, train_err: 99.0077, val_err: 50.9813
epoch: 164, train_err: 98.9807, val_err: 50.9665
epoch: 165, train_err: 98.8133, val_err: 50.7323
epoch: 166, train_err: 98.8024, val_err: 50.6888
epoch: 167, train_err: 98.7918, val_err: 50.6457
epoch: 168, train_err: 98.6432, val_err: 50.4348
epoch: 169, train_err: 98.5077, val_err: 50.2392
epoch: 170, train_err: 98.5014, val_err: 50.2011
epoch: 171, train_err: 98.3327, val_err: 50.3734
epoch: 172, train_err: 98.2958, val_err: 50.3677
epoch: 173, train_err: 98.2647, val_err: 50.3627
epoch: 174, train_err: 98.2543, val_err: 50.3202
epoch: 175, train_err: 98.2266, val_err: 50.3172
epoch: 176, train_err: 98.2033, val_err: 50.3144
epoch: 177, train_err: 98.184, val_err: 50.3119
epoch: 178, train_err: 98.1705, val_err: 50.2673
epoch: 179, train_err: 98.1573, val_err: 50.2233
epoch: 180, train_err: 98.0402, val_err: 50.4007
epoch: 181, train_err: 98.0251, val_err: 50.3541
epoch: 182, train_err: 97.9273, val_err: 50.5275
epoch: 183, train_err: 97.9103, val_err: 50.4786
epoch: 184, train_err: 97.7195, val_err: 50.2281
epoch: 185, train_err: 97.5469, val_err: 49.9964
epoch: 186, train_err: 97.5358, val_err: 49.9535
epoch: 187, train_err: 97.5251, val_err: 49.9111
epoch: 188, train_err: 97.5084, val_err: 49.9102
epoch: 189, train_err: 97.4969, val_err: 49.8675
epoch: 190, train_err: 97.3362, val_err: 49.6557
epoch: 191, train_err: 97.3273, val_err: 49.6158
epoch: 192, train_err: 97.1821, val_err: 49.4223
epoch: 193, train_err: 97.0492, val_err: 49.243
epoch: 194, train_err: 97.0091, val_err: 49.2561
epoch: 195, train_err: 96.8616, val_err: 49.4346
epoch: 196, train_err: 96.8535, val_err: 49.3953
epoch: 197, train_err: 96.7153, val_err: 49.2057
epoch: 198, train_err: 96.5784, val_err: 49.3815
epoch: 199, train_err: 96.5705, val_err: 49.3417
epoch: 200, train_err: 96.5564, val_err: 49.3428
```

```
figure(), grid, hold on
plot(t_err, 'b'), plot(v_err, 'r')
legend('Training error', 'Validation error')
title('Mini-batch training on IRIS dataset')
```



Of course, you could implement an online learning, enable the weight-decay, update them using momentum, apply an early-stopping criterion, or create an arbitrary error function, as discussed in the previous sections.

A test on a more complex dataset: MNIST

Consider now the dataset MNIST, containing images of hand-written digits. Consider the first 3000 elements for the training set, and the next 1000 for the validation set.

```
clear variables
X = loadMNISTImages('mnist/train-images-idx3-ubyte');
L = loadMNISTLabels('mnist/train-labels-idx1-ubyte');
T = getTargetsFromLabels(L);
X_t = X(:,1:3000)';
T_t = T(:,1:3000)';
X_v = X(:,3001:4000)';
T_v = T(:,3001:4000)';
```

Now, build a network with 784 inputs (the number of pixel in each image), 50 internal nodes and 10 output nodes.

```
net = Network(784, [50 10]);
net.name = "mnist_net";
```

The learning is executed for 200 epochs, with a batch approach and learning rate $\eta = 0.0005$. Training and validation error are plotted.

```
[t_err, v_err, best_net] = net.learn(X_t, T_t, X_v, T_v, 200, 0.0005);
```

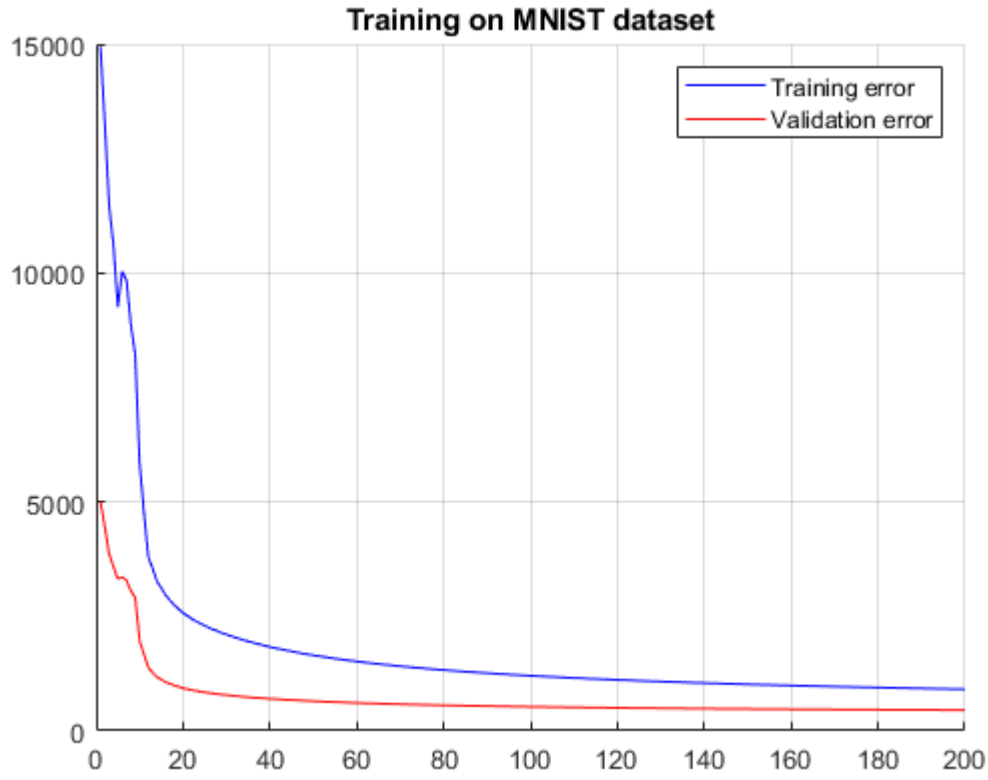
```
init err: 16320.8863
epoch: 1, train_err: 14946.1134, val_err: 5007.6474
epoch: 2, train_err: 13270.8445, val_err: 4460.6355
epoch: 3, train_err: 11454.7651, val_err: 3859.6048
epoch: 4, train_err: 10607.9011, val_err: 3580.3187
epoch: 5, train_err: 9255.3776, val_err: 3319.1368
epoch: 6, train_err: 10034.7113, val_err: 3360.3707
epoch: 7, train_err: 9837.3288, val_err: 3289.0347
epoch: 8, train_err: 8874.8593, val_err: 3053.7447
epoch: 9, train_err: 8223.125, val_err: 2912.998
epoch: 10, train_err: 5861.4004, val_err: 1978.0482
epoch: 11, train_err: 4782.3437, val_err: 1671.1788
epoch: 12, train_err: 3806.2182, val_err: 1386.1488
epoch: 13, train_err: 3543.2146, val_err: 1267.8903
epoch: 14, train_err: 3263.8531, val_err: 1175.1572
epoch: 15, train_err: 3122.6479, val_err: 1119.5528
epoch: 16, train_err: 2962.0912, val_err: 1065.8464
epoch: 17, train_err: 2855.2312, val_err: 1030.7937
epoch: 18, train_err: 2741.3114, val_err: 986.7318
epoch: 19, train_err: 2661.0764, val_err: 964.6742
epoch: 20, train_err: 2576.0055, val_err: 929.7031
epoch: 21, train_err: 2513.5209, val_err: 914.372
epoch: 22, train_err: 2447.3107, val_err: 887.222
epoch: 23, train_err: 2395.7233, val_err: 874.8474
epoch: 24, train_err: 2342.2067, val_err: 853.87
epoch: 25, train_err: 2297.5471, val_err: 842.7616
epoch: 26, train_err: 2252.6361, val_err: 826.2344
epoch: 27, train_err: 2212.887, val_err: 815.8028
epoch: 28, train_err: 2173.9638, val_err: 802.3775
epoch: 29, train_err: 2138.1278, val_err: 792.5127
epoch: 30, train_err: 2103.5694, val_err: 781.2637
epoch: 31, train_err: 2071.0448, val_err: 772.017
epoch: 32, train_err: 2039.8665, val_err: 762.3176
epoch: 33, train_err: 2010.2025, val_err: 753.7437
epoch: 34, train_err: 1981.7959, val_err: 745.1771
epoch: 35, train_err: 1954.6276, val_err: 737.2788
epoch: 36, train_err: 1928.5865, val_err: 729.5746
epoch: 37, train_err: 1903.6108, val_err: 722.3101
epoch: 38, train_err: 1879.6336, val_err: 715.293
epoch: 39, train_err: 1856.5925, val_err: 708.5996
epoch: 40, train_err: 1834.4338, val_err: 702.1517
epoch: 41, train_err: 1813.1026, val_err: 695.9642
epoch: 42, train_err: 1792.5514, val_err: 690.0021
epoch: 43, train_err: 1772.7332, val_err: 684.261
epoch: 44, train_err: 1753.6058, val_err: 678.7219
epoch: 45, train_err: 1735.1289, val_err: 673.3758
epoch: 46, train_err: 1717.2656, val_err: 668.2099
epoch: 47, train_err: 1699.9815, val_err: 663.2151
epoch: 48, train_err: 1683.2451, val_err: 658.3818
epoch: 49, train_err: 1667.0269, val_err: 653.7019
epoch: 50, train_err: 1651.3, val_err: 649.1677
epoch: 51, train_err: 1636.0396, val_err: 644.7723
epoch: 52, train_err: 1621.2228, val_err: 640.5095
epoch: 53, train_err: 1606.8286, val_err: 636.3737
epoch: 54, train_err: 1592.8378, val_err: 632.3597
epoch: 55, train_err: 1579.2325, val_err: 628.4628
epoch: 56, train_err: 1565.9964, val_err: 624.6786
epoch: 57, train_err: 1553.1141, val_err: 621.0031
```

```
epoch: 58, train_err: 1540.5714, val_err: 617.4326
epoch: 59, train_err: 1528.3548, val_err: 613.9633
epoch: 60, train_err: 1516.4516, val_err: 610.592
epoch: 61, train_err: 1504.8497, val_err: 607.3152
epoch: 62, train_err: 1493.5376, val_err: 604.1298
epoch: 63, train_err: 1482.5042, val_err: 601.0326
epoch: 64, train_err: 1471.739, val_err: 598.0207
epoch: 65, train_err: 1461.232, val_err: 595.0909
epoch: 66, train_err: 1450.9735, val_err: 592.2404
epoch: 67, train_err: 1440.9544, val_err: 589.4664
epoch: 68, train_err: 1431.166, val_err: 586.766
epoch: 69, train_err: 1421.6001, val_err: 584.1365
epoch: 70, train_err: 1412.2487, val_err: 581.5753
epoch: 71, train_err: 1403.1043, val_err: 579.0798
epoch: 72, train_err: 1394.16, val_err: 576.6475
epoch: 73, train_err: 1385.4088, val_err: 574.276
epoch: 74, train_err: 1376.8444, val_err: 571.963
epoch: 75, train_err: 1368.4606, val_err: 569.7062
epoch: 76, train_err: 1360.2518, val_err: 567.5034
epoch: 77, train_err: 1352.2122, val_err: 565.3525
epoch: 78, train_err: 1344.3367, val_err: 563.2517
epoch: 79, train_err: 1336.6202, val_err: 561.1988
epoch: 80, train_err: 1329.0578, val_err: 559.1921
epoch: 81, train_err: 1321.6449, val_err: 557.2299
epoch: 82, train_err: 1314.3771, val_err: 555.3103
epoch: 83, train_err: 1307.25, val_err: 553.4319
epoch: 84, train_err: 1300.2597, val_err: 551.5929
epoch: 85, train_err: 1293.4019, val_err: 549.792
epoch: 86, train_err: 1286.6731, val_err: 548.0277
epoch: 87, train_err: 1280.0694, val_err: 546.2986
epoch: 88, train_err: 1273.5872, val_err: 544.6033
epoch: 89, train_err: 1267.2232, val_err: 542.9407
epoch: 90, train_err: 1260.974, val_err: 541.3096
epoch: 91, train_err: 1254.8364, val_err: 539.7087
epoch: 92, train_err: 1248.8072, val_err: 538.137
epoch: 93, train_err: 1242.8834, val_err: 536.5935
epoch: 94, train_err: 1237.0621, val_err: 535.0772
epoch: 95, train_err: 1231.3404, val_err: 533.5871
epoch: 96, train_err: 1225.7157, val_err: 532.1223
epoch: 97, train_err: 1220.1853, val_err: 530.6822
epoch: 98, train_err: 1214.7465, val_err: 529.2657
epoch: 99, train_err: 1209.397, val_err: 527.8723
epoch: 100, train_err: 1204.1343, val_err: 526.5013
epoch: 101, train_err: 1198.956, val_err: 525.152
epoch: 102, train_err: 1193.86, val_err: 523.8237
epoch: 103, train_err: 1188.8439, val_err: 522.516
epoch: 104, train_err: 1183.9058, val_err: 521.2284
epoch: 105, train_err: 1179.0435, val_err: 519.9603
epoch: 106, train_err: 1174.2551, val_err: 518.7113
epoch: 107, train_err: 1169.5388, val_err: 517.4811
epoch: 108, train_err: 1164.8926, val_err: 516.2691
epoch: 109, train_err: 1160.3147, val_err: 515.075
epoch: 110, train_err: 1155.8036, val_err: 513.8986
epoch: 111, train_err: 1151.3576, val_err: 512.7395
epoch: 112, train_err: 1146.975, val_err: 511.5973
epoch: 113, train_err: 1142.6544, val_err: 510.4719
epoch: 114, train_err: 1138.3943, val_err: 509.3629
epoch: 115, train_err: 1134.1933, val_err: 508.2701
epoch: 116, train_err: 1130.05, val_err: 507.1932
epoch: 117, train_err: 1125.9632, val_err: 506.1321
epoch: 118, train_err: 1121.9317, val_err: 505.0865
epoch: 119, train_err: 1117.9542, val_err: 504.0562
epoch: 120, train_err: 1114.0295, val_err: 503.041
epoch: 121, train_err: 1110.1566, val_err: 502.0406
```

```
epoch: 122, train_err: 1106.3343, val_err: 501.0549
epoch: 123, train_err: 1102.5617, val_err: 500.0836
epoch: 124, train_err: 1098.8377, val_err: 499.1266
epoch: 125, train_err: 1095.1613, val_err: 498.1836
epoch: 126, train_err: 1091.5317, val_err: 497.2545
epoch: 127, train_err: 1087.9479, val_err: 496.3391
epoch: 128, train_err: 1084.409, val_err: 495.4371
epoch: 129, train_err: 1080.9141, val_err: 494.5483
epoch: 130, train_err: 1077.4625, val_err: 493.6726
epoch: 131, train_err: 1074.0533, val_err: 492.8098
epoch: 132, train_err: 1070.6857, val_err: 491.9596
epoch: 133, train_err: 1067.359, val_err: 491.1218
epoch: 134, train_err: 1064.0723, val_err: 490.2963
epoch: 135, train_err: 1060.8251, val_err: 489.4829
epoch: 136, train_err: 1057.6165, val_err: 488.6813
epoch: 137, train_err: 1054.4458, val_err: 487.8913
epoch: 138, train_err: 1051.3124, val_err: 487.1128
epoch: 139, train_err: 1048.2157, val_err: 486.3455
epoch: 140, train_err: 1045.1549, val_err: 485.5893
epoch: 141, train_err: 1042.1294, val_err: 484.844
epoch: 142, train_err: 1039.1387, val_err: 484.1093
epoch: 143, train_err: 1036.182, val_err: 483.3851
epoch: 144, train_err: 1033.2589, val_err: 482.6712
epoch: 145, train_err: 1030.3687, val_err: 481.9674
epoch: 146, train_err: 1027.5108, val_err: 481.2735
epoch: 147, train_err: 1024.6848, val_err: 480.5893
epoch: 148, train_err: 1021.89, val_err: 479.9147
epoch: 149, train_err: 1019.1259, val_err: 479.2494
epoch: 150, train_err: 1016.392, val_err: 478.5933
epoch: 151, train_err: 1013.6878, val_err: 477.9462
epoch: 152, train_err: 1011.0127, val_err: 477.3079
epoch: 153, train_err: 1008.3663, val_err: 476.6783
epoch: 154, train_err: 1005.7481, val_err: 476.0571
epoch: 155, train_err: 1003.1576, val_err: 475.4443
epoch: 156, train_err: 1000.5944, val_err: 474.8395
epoch: 157, train_err: 998.0579, val_err: 474.2428
epoch: 158, train_err: 995.5477, val_err: 473.6539
epoch: 159, train_err: 993.0634, val_err: 473.0726
epoch: 160, train_err: 990.6045, val_err: 472.4988
epoch: 161, train_err: 988.1707, val_err: 471.9324
epoch: 162, train_err: 985.7614, val_err: 471.3731
epoch: 163, train_err: 983.3763, val_err: 470.8209
epoch: 164, train_err: 981.015, val_err: 470.2756
epoch: 165, train_err: 978.677, val_err: 469.7371
epoch: 166, train_err: 976.362, val_err: 469.2051
epoch: 167, train_err: 974.0696, val_err: 468.6797
epoch: 168, train_err: 971.7995, val_err: 468.1606
epoch: 169, train_err: 969.5512, val_err: 467.6476
epoch: 170, train_err: 967.3245, val_err: 467.1408
epoch: 171, train_err: 965.1189, val_err: 466.6399
epoch: 172, train_err: 962.9341, val_err: 466.1449
epoch: 173, train_err: 960.7699, val_err: 465.6556
epoch: 174, train_err: 958.6258, val_err: 465.1718
epoch: 175, train_err: 956.5015, val_err: 464.6936
epoch: 176, train_err: 954.3968, val_err: 464.2207
epoch: 177, train_err: 952.3114, val_err: 463.7531
epoch: 178, train_err: 950.2449, val_err: 463.2906
epoch: 179, train_err: 948.1971, val_err: 462.8332
epoch: 180, train_err: 946.1676, val_err: 462.3807
epoch: 181, train_err: 944.1563, val_err: 461.9331
epoch: 182, train_err: 942.1629, val_err: 461.4902
epoch: 183, train_err: 940.187, val_err: 461.052
epoch: 184, train_err: 938.2285, val_err: 460.6184
epoch: 185, train_err: 936.2871, val_err: 460.1893
```

```
epoch: 186, train_err: 934.3625, val_err: 459.7645
epoch: 187, train_err: 932.4546, val_err: 459.344
epoch: 188, train_err: 930.5631, val_err: 458.9278
epoch: 189, train_err: 928.6877, val_err: 458.5157
epoch: 190, train_err: 926.8284, val_err: 458.1077
epoch: 191, train_err: 924.9847, val_err: 457.7037
epoch: 192, train_err: 923.1566, val_err: 457.3035
epoch: 193, train_err: 921.3439, val_err: 456.9072
epoch: 194, train_err: 919.5462, val_err: 456.5147
epoch: 195, train_err: 917.7635, val_err: 456.1259
epoch: 196, train_err: 915.9956, val_err: 455.7407
epoch: 197, train_err: 914.2421, val_err: 455.359
epoch: 198, train_err: 912.503, val_err: 454.9808
epoch: 199, train_err: 910.7781, val_err: 454.6061
epoch: 200, train_err: 909.0671, val_err: 454.2347
```

```
figure(), grid, hold on
plot(t_err, 'b'), plot(v_err, 'r')
legend('Training error', 'Validation error')
title('Training on MNIST dataset')
```



Implementing a convolutional network

Similarly to the shallow network, the architecture of the convolutional network has been implemented with two classes: *ConvLayer* and *ConvNetwork*.

The **ConvLayer** class represents one convolutive layer of a deep network. The layer is made of up a number of feature maps and max pooling layers: the first ones are object of *Layer* type, while the latter got a proper implementation as an operation performed on the output of feature maps.

In order to create a convolutive layer, the following parameters are needed:

- dimension of inputs as a vector of three elements (volume)
- dimension of the filters as a vector of two elements (dimensions must be even, so that a center pixel can be identified)
- number of feature maps
- dimensions of max pooling windows as a vector of two elements

```
% Creating a convolutional layer that accepts as input an image from MNIST,
% uses 3x3 filters, has 5 feature maps and 2x2 max pooling windows

MyConvLayer = ConvLayer([28,28,1], [3,3], 5, [2,2]);
```

The **compute** function computes the output of the layer given the input, which dimensions must be coherent with the layer. The function returns three values: the first one is the output of the convolutive layer, as a three dimensional matrix (the third dimension is equal to the number of feature maps); the second one is a cell array containing the outputs of the feature maps; the third one containts the indices of the elements selected by the max pooling.

```
% Example
[ConvLayerOutput, FeatureMapsOutput] = compute(MyConvLayer, randn(28,28,1));
```

The **ConvNetwork** class represents a convolutional network: it accepts a volumetric input, and contains any number of layers, each one with any number of feature maps, and with filters of any dimensions.

In order to create a *ConvNetwork* you need to specify the dimensions of input and output, the number of layers and feature maps, and the dimensions of the filters.

For example, in order to create a network that accepts a 30x30 pixels RGB image (3 channels), 4 outputs, two layers with 10 and 5 feature maps respectively, and 3x3 filters:

```
net = ConvNetwork([30, 30, 3], 4, [10, 5], [3, 3]);
```

ConvNetwork has the same structure as *Network;* you can specify its name, define a new error function or use the ones already implemented in the library - cross entropy with softmax (default) or sum of squares. The learning function's synax is the same too, so you can specify:

- training and validation set
- number of epochs
- weight update algorith, like R-prop (default) or gradient descent and its hyperparameters
- batch approach (default), mini-batch (of any dimensions) or online
- enabling or disabling weight-decay
- enabling or disabling weight update using momentum

- an early stopping criterion: *"firstMin"* or *"stopEpochs"*, like defined in the section regarding the *Network* class

Since the convolutional network is the same structure of the shallow network, there is no example here on how to use it; instead, the problem of classification of MNIST images is tackled in the next section.

Classifying hand-written digits (MNIST) using a convolutional network

Consider the classification problem of the *MNIST* dataset, containing images of hand-written digits using a convolutional network. These are gray-scale 28x28 image, thus defined on a single channel, classified in 10 classes of course.

The *test.m* script loads the dataset, initializes the network, trains it and memorizes the resulting accuracy. At the beginning of the script, you can find all the parameters that define the network's behaviour: in this way, it is straightforward to modify the values you need and execute the script.

Choosing the hyperparameters

In order to test the network, different set of parameters has been used and the different outputs compared. A value of $\eta = [0.5,\ 1.02]$ is capable of giving a sufficiently rapid convergence, as well as avoiding local minima and numerical instability.

At this point, a *"grid"* approach has been used, in order to have a complete and exhaustive vision of the network behaviour while varying its parameters. In particular, regarding a single convolutive layer, we tested the use of 5, 10, 15, and 20 feature maps, and each test has been repetead with filters of dimensions 3x3, 5x5, 7x7 e 9x9.

Because of the long time needed for the learning stage, each test has been executed only once, and their results saved on files in the *results* directory; also, these tests have been executed on a limited number of data and epochs. The networks that gave the best results have been trained several times using much more data and epochs, in order to obtain different accuracy values: average and standard deviation have been computed.

Tests with a single convolutive layer

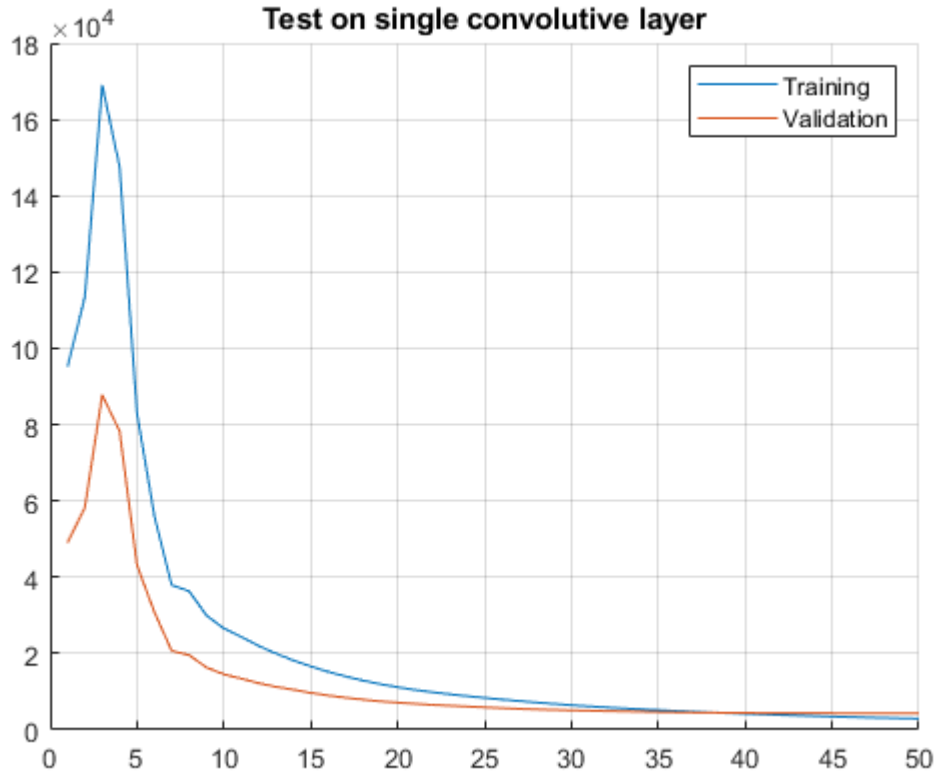
The two most promising architectures of the network with only one convolutive layer are the one with 10 feature maps and 3x3 filters, and the one with 15 feature maps and 5x5 filters. These networks have been trained on a training set containing 1000 elements, while validation set and test set consist of 500 elements.

The results of the test of the first network are loaded, the accuracy is computed, and training and validation error are plotted.

```
load('results/10map_filterDim3x3_LONG(1)');

% Plot
```

```
figure(), grid, hold on
plot(terr), plot(verr)
legend('Training', 'Validation')
title('Test on single convolutive layer')
```




```
% Accuracy
disp(['exact: ', num2str(exact), '; total: ', num2str(testDim), '; accuracy: ', num2str(accurac
```

```
exact: 389; total: 500; accuracy: 0.778
```

Due to the long time needed for the program execution, only three different measurements have been realized. The three accuracies are loaded, and their average and variance are computed:

```
testNum = 3;                % Number of tests
meas = zeros(1,testNum);    % Measurements
for i = 1 : testNum
    % Load only the accuracy
    load(['results/10map_filterDim3x3_LONG(', num2str(i), ')'], 'accuracy');
    meas(i) = accuracy;
end
disp(['Accuracies: ', num2str(meas)]);
```

```
Accuracies: 0.778       0.81       0.798
```

```
disp(['Mean: ', num2str(mean(meas))]);
```

```
Mean: 0.79533
```

```
disp(['Std: ', num2str(std(meas))]);
```

```
Std: 0.016166
```

The same is done for the second network:

```
testNum = 3;                % Number of tests
meas = zeros(1,testNum);    % Measurements
for i = 1 : testNum
    % Load only the accuracy
    load(['results/15map_filterDim5x5_LONG(', num2str(i), ')'], 'accuracy');
    meas(i) = accuracy;
end
disp(['Accuracies: ', num2str(meas)]);
```

```
Accuracies: 0.808       0.804       0.83
```

```
disp(['Mean: ', num2str(mean(meas))]);
```

```
Mean: 0.814
```

```
disp(['Std: ', num2str(std(meas))]);
```

```
Std: 0.014
```

The second network got better performances than the first one: in both case, however, results are consistent (standard deviation is small) and satisfying, considering that the network has been trained on a small training set.

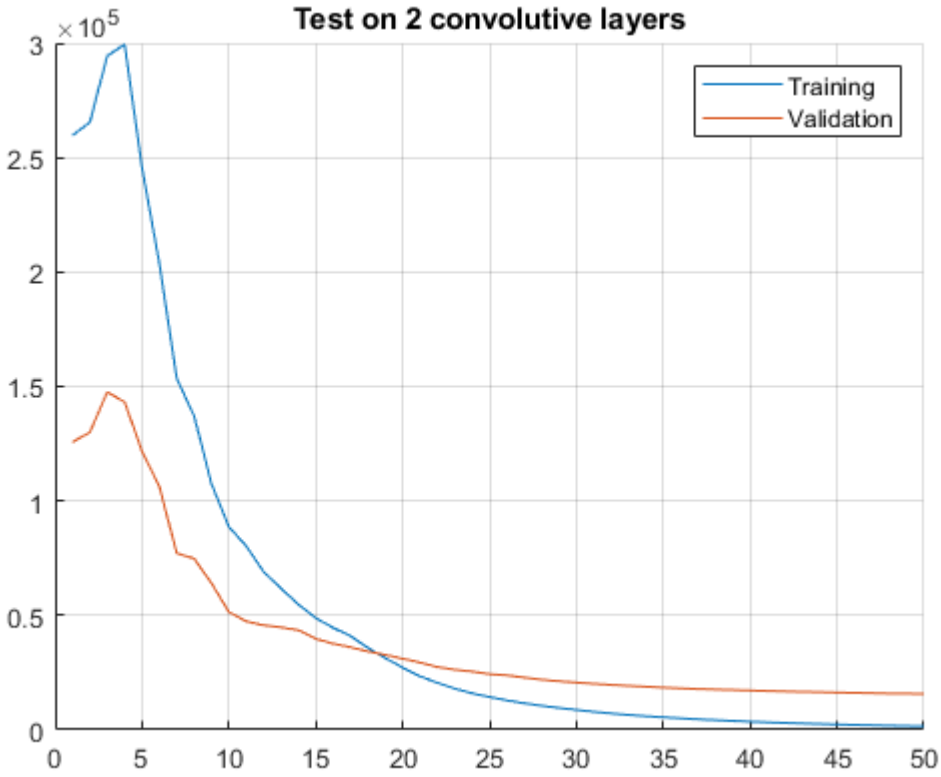
Tests with more convolutive layers

Networks with more convolutive layers has been considered too. Due to computational reasons, even smaller datasets has been used: only 500 elements for the training set, 250 elements for validation and test sets.

The results are much worse than the previous model with just one layer: however, an investigation on a much bigger dataset is needed before giving conclusions. The best results was given by a network with 2 layers, with 10 and 15 feature maps respectively, using 3x3 filters.

```
load('results/2layers_10_15map_filterDim3x3');

% Plot
figure(), grid, hold on
plot(terr), plot(verr)
legend('Training', 'Validation')
title('Test on 2 convolutive layers')
```




```
% Accuracy
disp(['exact: ', num2str(exact), '; total: ', num2str(testDim), '; accuracy: ', num2str(accurac
```

```
exact: 179; total: 250; accuracy: 0.716
```

Further improvements

The network developed in this project gave good results, but chances of further improvements and developments are clear, as well as the obtainable performances. In particular:

- train the networks on much bigger dataset, that have been severly limited by time and computational requirements. The whole *MNIST* dataset is composed by 60k elements, while in this project we used at maximum 2k elements.
- use filters of different dimensions for each layer, or a padding different from 1
- test different weight update algorithms (in this project, batch and rProp were required)