March 2014 - Machine Learning
Barberakis Stelios – AM: 2008030116
chefarov@gmail.com

Approaching Optical Character Recognition using MLPs and SVMs

1. Introduction

Optical character recognition is one of the older and most common task in Machine Learning. It as a convenient field for someone who wants to try learning techniques and pattern recognition methods on real-world data while spending minimal efforts on preprocessing and feature extraction (depends on the method used), since huge and well-formed (preprocessed) datasets exist like (MNIST/NIST).

Moreover OCR has some important practical applications like:

- Assisting blind and/or visually impaired people
- Automatic Postcode recognition
- Convert scanned documents to text files which is the first step for a variety of digital applications. [2]

In this project we are actually classifying digits, not letters, though the topics and challenges encountered are common for both problems, thus we are going to refer to **OCR (for convenience) instead of 'handwritten digit recognition' since now.**

Lastly we should mention that there are some steps before encountering OCR when we want to convert scanned text to digital text. The digitization of a scanned document is followed by reducing noise techniques and then a proper segmentation is applied in order to extract the lines, then the words and finally the discrete characters are extracted. However there are approaches when classification is done directly on word-level (e.g without even extracting characters), [1] [2]. Last but not least it's worth mentioned that human eye also works on a word-level language model.

1.1. Dataset Used

In this project we are trying make our computer able to classify handwritten or scanned images of numbers, as number: 0 to 9. In order to train our classifiers we use the MNIST dataset, which is

considered the most standard (coherent and large enough) dataset of handwritten numbers. It is frequently used in order to compare classifiers or report new image pattern recognition techniques

MNIST consists of 70,000 images (28x28 pixels), where each picture represents **a single digit**. The first 60,000 pictures are the training set, and the rest 10,000 consist the testing set. The testing set is formed with digits written by people that didn't participate to the construction of the training set. [3]

2. Related Work

There are two basic approaches to solve image classification problems:
Operating at the input space (image level)
Operating at the feature space (extract features from the imagery)

A lot of the OCR approaches include some mild preprocessing on the data (e.g deskewing), following an exhaustive search to calculate the optimal parameters of a classifier (1st category above). [3]

On the other hand there are a lot of feature extraction models encountered, which mainly reside in two general categories:
Shape descriptors (structural analysis)
Image deformation models

Image deformation stands for operations that perform various transformations at the image in order to succeed some form of alignment between them or to describe each instance based on geometric characteristics (often shape descriptors). [4]. A popular transformation for the OCR problem is the affine transformation [5] [6]

Popular shape descriptors include the Fourier Coefficients method, the shape context, e.g. Shape is invariant to some distortions like scaling, translating, rotating of the image, that can be seen even to characters coming from the same source (e.g person), which is the main reason behind misclassifications.

Some state of the art implementation's error ranges reported in [3]:

MNIST test error rate (best values reported):

- Linear Classifiers: 7.6 – 12%
- kNN: 0.63 – 5%
- Boosted Stumps: 0.87 – 7.7%
- PCA with Quadratic Classifiers: 3.3%
- SVM: 0.56 – 1.4%
- ANN (e.g. MLP, CNN): 0.23 – 4.7%
- MCDNN: 0.23%
- Human error: 0.2%

2.1 Best Solution

The best solution so far is the Multicolumn Deep Neural Network (MCDNN) implementation by Dan Cires¸an, Ueli Meier and J¨urgen Schmidhuber. MCDNN is a Convolution (or Cellular) neural network (CNN). [10]

2.1.1 Cellular Networks in general

CNNs are a hierarchical variant of MLPs, inspired by neurobiology. The interface between symbolic data manipulated in high-level vision and signals processed by low-level operations is considered one of the major problems in Computer Vision, and the main reason human performance in visual perception by far exceeds the performance of con-temporary computer vision systems [7]

CNNs try to exploit the above problem by working in a hierarchical manner, where lower level layers check hypothesis made on higher layers by producing simple features, that are used by higher layers to produce more complex features [8]

Their structure follows (abstractly) the the one between retina and visual cortex. Input space is divided in small chunks called receptive fields (similar idea behind RBF networks). Winer-take-all neuron results tile in each layer in a way that they cover the entire virtual space. This way locality correlation is achieved, making CNNs invariant to translation of the input (image) [9]. A CNN consists of multiple layers that include multiple maps.

CNNs are considered the state-of-the art solution for image classification problems. Expert knowledge is not needed in order to extract meaningfull features, as the network through its hierarchical structure, produces it itself [8].

2.1.2 MCDNN (2012) Implementation

The classifier used by [10], achieves an 0.23 test error on MNIST

dataset (human error is 0.20). The description of the DNN architecture is: 1x29x29-20C4-MP2-40C5-MP3-150N-10N, that represents a net with 1 input image of size 28x28, a convolutional layer with 20 maps and 4x4 filter, a max-pooling layer over non overlapping regions of size 2x2, a convolutional layer with 40 maps and 5x5 filter, a max-pooling layer over non overlapping regions of size 3x3, a fully connected layer with 150 hidden units, a fully connected layer with 10 hidden units and a fully connected output layer with 10 neurons (one per class – softmax used). [11]

Copying from their publication:

The network consists of 2-dimensional layers of winner-take-all neurons with overlapping recep-tive fields whose weights are shared. Given some input pattern, a simple max pooling technique determines winning neurons by partitioning layers into quadratic regions of local inhibition, selecting the most active neuron of each region. The winners of some layer represent a smaller, down-sampled layer with lower resolution, feeding the next layer in the hierarchy

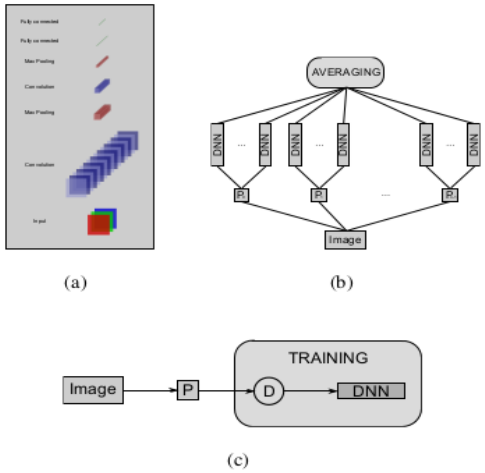


Figure 1. (a) DNN architecture. (b) MCDNN architecture. The input image can be preprocessed by $P_0 - P_{n-1}$ blocks. An arbitrary number of columns can be trained on inputs preprocessed in different ways. The final predictions are obtained by averaging individual predictions of each DNN. (c) Training a DNN. The dataset is preprocessed before training, then, at the beginning of every epoch, the images are distorted (D block). See text for more explanations.

The network is trained using online gradient descent (with anneal learning rate), which is a much costly training function, thus a network of GPUs is used in order to increase performance. [11]

3. Our Implementation

My motivation behind this project, was that a robust and preprocessed dataset with reported results to compare is available, something that gives me (and other beginners) more time to better understand the machine learning concept instead of getting familiar with the domain knowledge and applying several preprocessing steps.

In this project, a theoretical mostly overview of various classifiers is presented along with the steps I took to apply learning on MNIST dataset, something that hopefully may be usefull for other students as well. ML methods utilized include: MLP(with various architectures), simple kNN and SVMs.

I am using Python to implement all of the classifiers as is explained below.

3.1 Preprocessing

Each digit in MNIST dataset is aligned to the center by mass, meaning that the center of mass of the pixels has been calculated and that point is positioned in the center of 28x28 window [1]. Following the above process each digit is left with a padding of 4 black (background) pixels around it, which are invariant features, thus the preprocess script removes them, getting a 20x20 image. Moreover the 20x20 image is resized to 14x14, slightly reducing the large input space from 70000x20x20 to 70000x14x14. Finally before each classifier is called, our data is being reshaped to 70000x196, which is considered a proper input for most ML libraries, where each sample is represented by each row of the matrix.

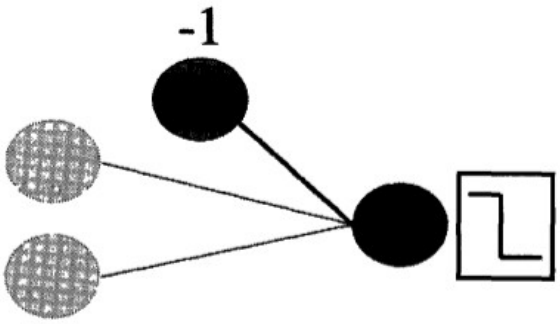
Experiments using the 10% of the dataset gave the same (some times slightly worse, some times slightly better) performance as the initial dataset (28x28), indicating that the above procedure didn't harm the overall accuracy of the model.

3.1 Multilayer Perceptron Network (MLP)

3.1.1 Perceptron.

Perceptron is a binary classifier that consist of one layer (output nodes) that are (every one of them) connected with all the input nodes (aka: the feature vector). Thus the number of weights is equal to the dimensions of the input (N) times the number of the output nodes (O).

The simplest perceptron, for a problem with two-dimensional input and one output is:



where the simplest activation function (output) is:

$$f(x) = \begin{cases} 1 & \text{if } w \cdot x + b > 0 \\ 0 & \text{otherwise} \end{cases}$$

Perceptron's output is an inner product of the input, thus a linear combination of the input. That's the reason that it can discriminate only linear seperable data. [12]

3.1.2 Multilayer Perceptron Network overview

Multilayer Perceptron Network is a perceptron network with at least on hidden layer, something that makes it capable of solving non linear problems, in contrast with the simple perceptron algorithm. It is a feed-forward fully connected network, meaning that information goes one direction, and there exist no loops. That is the difference with recurrent networks where output nodes are connected with the hidden layer, in perceptron more weights are added in order to solve more complex problems. [13]

MLP's can also be utilized in regression problems, and the unique thing about them is that a 3 layer net With sigmoidal activation functions can approximate any function to arbitrary accuracy: property of Universal Approximation [14]

The input presented in MLP must be in the format: MxN, where M is the number of samples in the dataset, and N is the number of their attributes (features). That's why we reshape each picture from 14x14

array to 196 element vector (N=196).

3.2.1 Training a MLP

There are two three modes of learning: sequential(online), batch and stochastic learning. Perceptron is a sequential learner, meaning that after each sample is presented, its weights are updated. MLP is natively a batch learning model since all of the samples are presented (multiplied by weights of the hidden layer), and after that the weights are updated. However MLP can also be trained sequentially something that speeds up the process in cost of error performance.

Learning in neural networks actually occurs by adjusting weights in order to decrease the classification/approximation error. Error is calculated usually as the euclidean distance between target and hypothesis(our output):

$$E(\mathbf{w}) = \frac{1}{2}\sum_{k=1}^{N}(t_k - y_k)^2$$

If we differentiate the above formula with respect to w, we get:

$$\frac{\partial E}{\partial w_{ik}} = \sum_{k}(t_k - y_k)(-x_i).$$

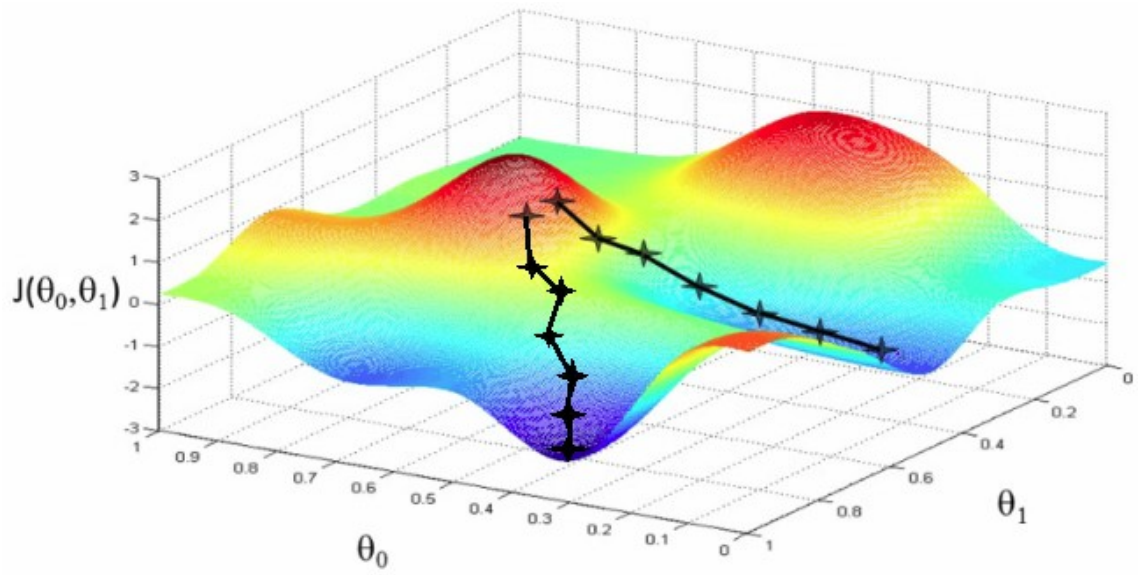
The idea behind gradient descent is that we follow the gradient downhill (minimize error), thus the weight update rule becomes:

$$w_{ik} \leftarrow w_{ik} + \eta(t_k - y_k)x_i,$$

where η is the learning factor. The bigger the η the faster our algorithm will converge to a local minimum. If we lower η we are getting a better downhill road, leading to a deeper minimum. Actually we are also use the momentum trick described in [16], hence our final update equation becomes:

$$w_{ij}^{t} \leftarrow w_{ij}^{t-1} + \eta\delta_o a_j^{\text{hidden}} + \alpha\Delta w_{ij}^{t-1},$$

The initialization of weights is stochastic (usually from a uniform distribution with values very close to zero). Thus each time we run the same network on the same problem a different route is followed, leading to another minimum (solution).



A mathematical overview of training procedure, taken from Marsland's book is the following [15]:

Forwards phase:

- compute the activation of each neuron j in the hidden layer(s) using:

$$h_j = \sum_i x_i v_{ij} \tag{3.4}$$

$$a_j = g(h_j) = \frac{1}{1+\exp(-\beta h_j)} \tag{3.5}$$

- work through the network until you get to the output layers, which have activations:

$$h_k = \sum_j a_j w_{jk} \tag{3.6}$$

$$y_k = g(h_k) = \frac{1}{1+\exp(-\beta h_k)} \tag{3.7}$$

Backwards phase:

- compute the error at the output using:

$$\delta_{ok} = (t_k - y_k)\, y_k (1 - y_k) \tag{3.8}$$

- compute the error in the hidden layer(s) using:

$$\delta_{hj} = a_j (1 - a_j) \sum_k w_{jk} \delta_{ok} \tag{3.9}$$

- update the output layer weights using:

$$w_{jk} \leftarrow w_{jk} + \eta \delta_{ok} a_j^{\text{hidden}} \tag{3.10}$$

- update the hidden layer weights using:

$$v_{ij} \leftarrow v_{ij} + \eta \delta_{hj} x_i \tag{3.11}$$

randomise the order of the input vectors so that you don't train in exactly the same order each iteration

The above algorithmic procedure is known as Back Propagation (BP). Gradient descent (GD) is known as the training function. Other training functions beside GD also exist. The following table is from matlab's neural network toolbox that contains several training functions for BP:

Training Functions	
trainbfg	BFGS quasi-Newton backpropagation.
trainbr	Bayesian regularization.
traincgb	Powell-Beale conjugate gradient backpropagation.
traincgf	Fletcher-Powell conjugate gradient backpropagation.
traincgp	Polak-Ribiere conjugate gradient backpropagation.
traingd	Gradient descent backpropagation.
traingda	Gradient descent with adaptive lr backpropagation.
traingdm	Gradient descent with momentum backpropagation.
traingdx	Gradient descent with momentum and adaptive lr backprop.
trainlm	Levenberg-Marquardt backpropagation.
trainoss	One-step secant backpropagation.
trainrp	Resilient backpropagation (Rprop).
trainscg	Scaled conjugate gradient backpropagation.
trainb	Batch training with weight and bias learning rules.
trainc	Cyclical order incremental training with learning functions.
trainr	Random order incremental training with learning functions.

In this project **Resilient Back Propagation** is also used. Rprop follows the same logic but computes update rules independently for each weight (+/-) following the partial derivatives change at each iteration.

3.2.2 Stopping

The known issue with the decision when to stop learning is that too much learning from the training data can lead to overfitting (learn the noise in the data), and the opposite situation is the underfitting phenomenon. To avoid overfitting one can utilize either Earlystopping procedure or L1,L2 regularization (extra term to the loss function like the baeysian approaches)

Early Stopping:

Early Stopping is a function that iteratively calls train function (for some epochs) and then calls the computes the validation set error in order to determine how well the network has generalized by that point.

Then we need a heuristic to decide when to stop. A simple solution is to stop if the validation error in last 2 or 3 (earlystopping) iterations has increasing. That way we decide that further learning leads to overfitting.

In this project the earlystopping technique is used. [16]

3.2.3 Architecture - Parameters

To conclude a MLP network is described by its architecture and the training parameters. These are the factors that affect the convergence point (thus the performance) that our network will reach.

- Architecture:
  - Number of layers (usually 1 or 2 are enough). Complexity grows exponentially with the number of layers
  - Number of nodes in each layer. More nodes (as well as more layers) make the model more flexible, thus more susceptible to overfitting phenomenon
  - Activation function of each node. We usually use sigmoid activation function in the hidden layers because as we saw in section 3.2.1, the error and thus the output must be differentiable. Output activation is usually:
    - Linear: for regression or
    - Softmax with 1-of-N encoding: for classification. This means that we have N-dim output where the N outputs sum to 1, thus representing a PDF. By that way we approximate better accuracy about how sure we are for the current prediction (what is the 2nd guess and so).

- Training parameters: Learning rate (η) and momentum(α) described in section 3.2.1

3.3 Implementation Steps

In this project I use the code provided by Stephen Marsland [17] as the Network Class in order to run Back-Propagation Gradient Descent (Batch type), and the pyBrain library in order to run Resilient Back-Propagation (Rprop has only sequential mode).

We prepare our data to be supplied to the MLP class, following the steps below:

1. We get mnist in three chunks from our preprocess script: test set, valid set, and test set (and the according targets).
2. Convert input (train, valid, test) from Mx14x14 to Mx196
3. Convert target (train,valid,test) from Mx1 to Mx10 following 1-of-N encoding using softmax.
4. Get a subset of the dataset if we want to investigate good parameters for our model instead of doing the final training
5. Normalize all of the inputs (train,valid,test) using the formula: x= (x – E[x]) / max(x). That way our trained model will be able to compare new uknown input of any intensity range
6. Transform our input to 'ClassificationDataSet' objects of the pybrain library, if we want to use pybrain (RPROP here).
7. Build the network by supplying the train set along with each targets. That way the MLP class knows what dimensions the weights matrices must have.
8. Call the training function giving the training parameters (iterations, learning rate, momentum).
9. Print the results

3.4 Results

The following table summarizes the most indicating results for 2 and 3 layer architectures from the **Gradient Descent** method.

Architecture (nodes)	Dataset usage %	Performance	Training time
50	50	95.60	14h 40m
50	100	96.83	33h 20m
100	100	97.44 (vs 95.3 of 300-nodes reported)	66h 0m
—----------	—----------	—----------	—----------
150-100	100	97.17	5days 15h
200-100	100	97.41	52h
300-100	100	97.64 (best reported: 97)	8days

The sensible choice would be the 2-layer network with 100 hidden nodes (3rd row), since it achieves 97.44% performance (2.66% test error), in 2days and 18h against the least better performance of 300-

100 architecture which took 8 days to train.

However as it is clear from the above table, the training time can have huge variation, depending on the stochastic initialization of the weights (check the incredible huge difference between 150-100 and 200-100 architectures that should have opposite execution times.

The R-Prop method from the pyBrain library initially achieved 5x or 10x faster training time for subsets of the dataset (10-20%), and decreased the error rate significantly. Applying a 2-layer (100 nodes) network gave as the unexpected result:

Performance: 99.15%, (aka: test error: 0.85%) running for 104h (much more than 66h for the according GD instance, despite the fact that Rprop is generally much faster).

That performance hasn't been reported (as far as I know) for a so small network without preprocessing. The ANN reported with similar performance is a “2-layer NN, 800 HU, [elastic distortions]”, which is apparently significantly more complex.

Using simple Back propagation (Gradient Descent), pyBrain network gave us a little more that 96% performance (close to our mlp.py) code, meaning that the performance achieved with Rprop is realistic (not a confusion issue in some step).

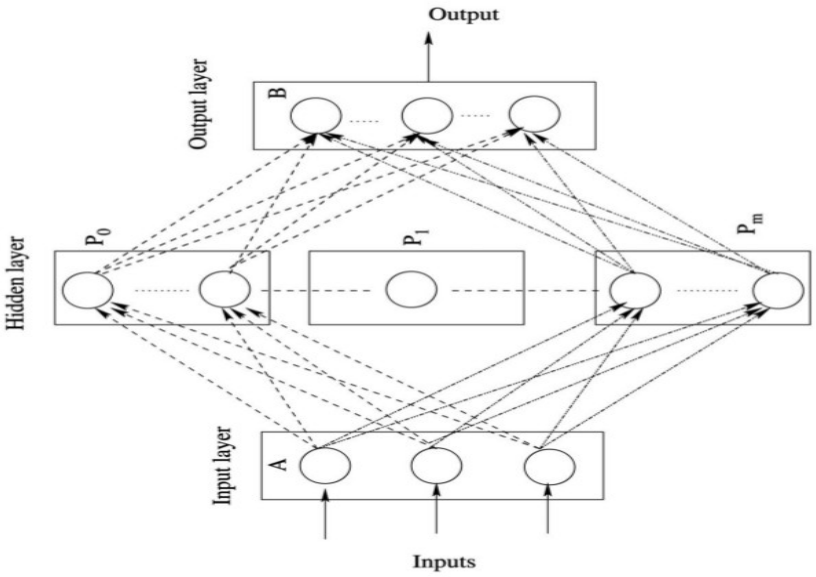
3.5 Optimization

Backpropagation methods obviously need parallelization to tune up their execution.

As conserns the sequential method, a thread pool where each worker (thread) computes the activation function for an individual neuron each time would be a sensible choice. Rprop falls in the same category

Batch training could be parallelized by splitting the dataset into buckets, where each thread computes the output for each bucket, and the results are summed.

A reported fast implementation is that of [18] where the calculation of hidden layers activations are divided among threads as shown below:



3.6 Saving the instance

A trained neural network is easily saved as it is a class object, through serialization to disk. In python we do that using the Pickle module.

A classify.py script was built in order to use the trained object and classify random pictures that we may wanted to try:

Input image:

gives the following execution:

```
chefarov@debian:~/programming/ML/ocr14$ python classify.py -p samples/7e.jpg -n NN_s_100_0.save
[[  0   0   0   0   0   0   0   0   0   0   0   0   0   0]
 [  0   0   0   0   0   0   0   0   0   0   0   1   0   0]
 [  0   0   0  10   5   2   2   2   2   2   2   0   0   0]
 [  0   1   0  37  36  31  38  42  48  57  54 101  16   0]
 [  0   2   0  72 153 142 151 155 128  99 109 199  18   0]
 [  0   0   0  14  28  24   6  44 164 160 197 124   0   4]
 [  0   0   0   0   0   0  98 153 209 220 133  14   1   1]
 [  0   0   0   1   0  82 214 199 219  70   0   0   0   0]
 [  0   1   2   6 134 197 175 205  73   0   5   1   0   0]
 [  0   2   0  95 202 179 191 155   0   4   0   0   0   0]
 [  1   0  36 169 173 169 207  66   1   3   0   0   0   0]
 [  1   0  39 167 171 180 186   7   0   1   0   0   0   0]
 [  0   0   0  18  20  19  14   2   0   0   0   0   0   0]
 [  0   0   1   0   0   0   0   0   0   0   0   0   0   0]]
[[  1.04696293e-04   1.19935424e-01   3.33057029e-04   1.86990869e-03
    3.31822519e-06   2.56551445e-08   1.47507321e-07   6.99006362e-01
    1.78744038e-01   2.87642129e-06]]
The number was classified as 7
```

3.7 Conclusion

Neural networks is a robust ML technique that can approximate any model given the proper preparation that must be followed after some reasoning taking in consideration the domain knowledge we have. Some pros/cons are listed:

(+) Invariant feature detection: Neural network's can identify the significant features (input attributes) by itself through its 1st layer, since at the end of the training the columns of Weights(1) matrix that have lower values correspond to less important features

(+) General applicability: They can be applied to regression, classification, time-series prediction, data compression (a.o), under several cirqumstances.

(+) Can approximate complex models: They have a large number of free variables, and with low learning factor we can use them as a kind of black-box solution (not encouraged to think in that manner though), when we haven't the appropriate domain knowledge and we don't have any other solution.

(-) A lot of experimentation needed: There isn't a methodology to choose network parameters, we have to run the network multiple times.

(-) Training phase too long, need for parallelization: Self explanatory. It would be important to mention though, that after 1000 first iterations (1 or 2h in our problem for 100% of the dataset) the performance increment is too small, meaning that we achieve a good approximation much earlier than we think.

(-) Difficult to make use of prior knowledge: As we mentioned There isn't a methodology to choose network parameters.

3.2 k-Nearest neighbors approach (kNN)

Nearest neighbors approaches are non-parametric methods classifying directly from the training set. A metric function must be defined (usually the euclidean distance) and is used on the feature space in order to compare the distance between a random sample with the rest of the training set. If we have a classification problem, that point is classified by the majority class-membership of its k neighbors. For example if k=1, the point is given the class membership of its nearest neighbor in the feature space.

Using a subset of the dataset we experimented with various values of k (1 to 14). Our code doesn't need mutliple executions, as k values are given in a list, thus a list with the corresponding classifiers is made.

K	accuracy 1st execution	accuracy 2nd execution	Average	Exec time (min)
1	91.7	92.6	92.15	4.7
2	90.9	92.2	91.6	9.5
3	92.2	93.4	92.8	14.79
4	91.8	92.3	92.05	18.7
5	92.1	92.5	92.3	23.5
6	91.7	92.6	92.15	27.9
7	91.9	92.9	92.4	32.7
8	92.5	93	92.5	39
14	91.5	92.4	91.9	

We found out that k=3 gave the best performance, something that was confirmed that was the case by searching the web. [19]

Using the entire dataset we got : 96.4% accuracy (3h 37m), working in input space (no feature extraction). Mnist webpage [3] reports 96.9% for the same model.

The most accurate nearest neighbor approaches (not exclusively) in image classification include some advanced machine vision techniques (mostly shape descriptors) as a metric function. Because knn is sensitive to data noise and even mild destortions of the input, these tasks are essential in order to make a more robust model with better generalization ability.

The best implementations reported (lower MNIST test error rate) are [3]:

✔ Shape Contexts (reported 99.37% performance). Shape context is a shape descriptor. A subset of the points consisting the contour are kept (eg: N) and the relevant positions for the rest of the points construct a histogram for the source point. Thus we conclude with N histograms for each shape, one for each point. A cost function is defined between a point i from one picture and a point j from another picture that depends on their histograms

$$C_{ij} \equiv C(p_i, q_j) = \frac{1}{2}\sum_{k=1}^{K}\frac{[h_i(k) - h_j(k)]^2}{h_i(k) + h_j(k)},$$

Considering the set of all costs between two pictures is the new metric function for our knn classifier. [20]

✔ Deformation models (reported: up to 99.48% performance). Check descriptions in the previous chapters.

3.2.3 Conclusion

+ Very simple, works suprisingly well in various scenarios

+ Easily implemented and pamaterized

+ Usefull for quick experiments

- Doesn't learn from the data. Repeats the N comparisons and sorting for each classification

- Sensitive to noisy data

- Doesn't generalize well

3.3 Support Vector Machines (SVM)

3.3.1 Overview

SVM is a binary classifier (it can be used for regression purposes as well).

SVM models include a kernel function that maps the input in a higher dimensional space in order to be linearly separable (in that dimension) by a hyperplane.

The goal of SVM is to produce a model (based on the training data) which predicts the target values of the test data given only the test data attributes

Given (xi,yi), i=1, . . . , l where xi ∈ R^n and y ∈ {1, −1}, the solution to the following optimization problem is needed:

$$\begin{aligned} \min_{\mathbf{w},b,\boldsymbol{\xi}} \quad & \frac{1}{2}\mathbf{w}^T\mathbf{w} + C\sum_{i=1}^{l}\xi_i \\ \text{subject to} \quad & y_i(\mathbf{w}^T\phi(\mathbf{x}_i) + b) \geq 1 - \xi_i, \\ & \xi_i \geq 0. \end{aligned}$$

This is the soft margin version of the problem, where we tolerate misclassified samples if a hyperplane that seperates all of the points to two classes is not possible. ξi represent the degree of misclassification for point I and C is the penalty term.

As the following equations imply, the parameters affecting our model is the choice for the C value as well the kernel (and its parameters).

The basic kernels for SVMs are the following:

- linear: $K(\mathbf{x}_i, \mathbf{x}_j) = \mathbf{x}_i^T\mathbf{x}_j$.
- polynomial: $K(\mathbf{x}_i, \mathbf{x}_j) = (\gamma {\mathbf{x}_i}^T\mathbf{x}_j + r)^d$, $\gamma > 0$.
- radial basis function (RBF): $K(\mathbf{x}_i, \mathbf{x}_j) = \exp(-\gamma\|\mathbf{x}_i - \mathbf{x}_j\|^2)$, $\gamma > 0$.
- sigmoid: $K(\mathbf{x}_i, \mathbf{x}_j) = \tanh(\gamma {\mathbf{x}_i}^T\mathbf{x}_j + r)$.

Linear kernels only when data linearly separable

Polynomial kernel is more complex because it has more parameters. Parameters: γ, r, d

RBF kernel is the most flexible model, transforming the input to higher dimensions. Parameters: γ

Generally a good first choice (for complex problems – lots of free variables) is usually the RBF kernel, followed by polynomial kernel.

3.3.2 Simple method - steps

>> Transform data to the format of an SVM package

>> Conduct simple scaling on the data

>> Consider the RBF kernel

>> Use cross-validation(or not) on a small subset to find a good neighborhood for (C,γ).

>> Make a finer grid search on the neighborhood to find better approximation for the optimal (C,γ)

>> Use the best parameter C and γ to train the whole training set

>> Evaluate generalization performance on the test set

In this project scikit-learn library is used which has a very friendly API in order to perform grid-search as well to fit the model and compute the test error. Moreover scikit-learn's svm library automatically identifies that our problem consists of more that 2 class labels, implementing the (one-versus-all) Multiclass model.

3.3.3 Performance:

Using 10% of the dataset, performed grid search, for poly and rbf kernel's parameters.

Indicative running times for 6-7 values for each parameter are:

RBF kernel: 50min, Polynomial kernel: 2h

Larger grids (10 values for each parameter) make polynomial model to run for many hours or more than a day

The optimal values for the RBF kernel are reported [21]:

>> RBF: (C,γ): C=2.8, γ=0.0073.

Using these parameters we achieved Performance: 98.6%, training time: 2h10m. That's the same performance like the MNIST page reports [3].

Polynomial kernel parameters are not reported in the publication cited by [3]. Our research gave as:

>> Poly (C,γ,ρ,d): C=0.35, ρ=0.125, γ=0.0625, d=3

Using these parameters we achieved Performance: 98,25%, training time: 14h. A 98.9% performance is reported in MNIST website (better approximation of parameters)

Best implementation: Virtual SVM, deg-9 poly, 2-pixel jittered achieved 99.44%

4. **Comparing:**

Our results show that for the OCR problem, working on the input space, MLP with resilient back propagation outperforms the other classifiers with SVM performance to be very close.

MLP has a long training phase but can easily even be disrupted after some iterations with good performance (we implemented a SIGKILL handler that saves the neural net even if the training is not over),

SVM is much much faster as concerns the training phase (some minutes), though the grid search for large grids for the polynomial kernel is costly similar to the MLP training (but still MLP must be re-run for other architectures as well).

5. Conclusion – Future work

Apparently OCR is a deeply researched issue, since we have human competitive results. The research efforts should probably be directed to the utilize character level classifiers like MDCNN in a complete framework that performs handwritten text to digital text conversions.

Personally I don't think I will deal with this issue further more.

I would like to make a parallel version of the Bprop code or the pybrain library because parallel MLP implementations are not available in the web (as far as I know there are only some wrappers for commercial frameworks like AngLib).

Check the execution instructions (another pdf) to run the code.

References

[1] Sukhpreet Singh: Optical Character Recognition Techniques: A Survey, Journal of Emerging Trends in Computing and Information Sciences, 2013, p. 4

[2] Pitrelli, J.F: Creating word-level language models for handwriting recognition, IEEE , 2001

[3] MNIST official website:

[4] Daniel Keysers, Thomas Deselaers, Christian Gollan, and Hermann Ney: Deformation Models for Image Recognition, IEEE 2007, v. 27

[5] Fabien Lauer a,∗ , Ching Y. Suen b and G´erard Bloch A trainable feature extractor for handwritten
digit recognition

[6] Affine transformation . Wikipedia

[7] Sven Behnke: Hierarchical Neural Networks for Image Interpretation. Phd Thesis. Springer-Verlag, 2013

[8] Sven 2013, p. 3

[9] Lecun: Lenet http://deeplearning.net/tutorial/lenet.html

[10] D. Cires, U. Meier, J. Schmidhuber: Multi-column Deep Neural Networks for Image Classification, IEEE 2012.

[11] Cires, Meier, Schmidhuber, p.2-3

[12] Stephen Marsland: Marchine Learning, An algorithmic perspective, CRC Press, ch. 2.2

[13] Marsland p. 47-50

[14] Wikipedia: Universal approximation theorem

[15] Marsland ch. 3.2

[16] Marsland ch. 3.3.6

[17] Stephen Marsland's personal Page

[18] S. Suresh, S.N. Omkar, and V. Mani: Parallel Implementation of Back-Propagation Algorithm in Networks of Workstations, IEEE 2005

[19] Ming Wu Zhen Zhang: Handwritten Digit Classification using the MNIST Data Set, p. 4

[20]Serge Belongie, Member, IEEE, Jitendra Malik, Member, IEEE, and Jan Puzicha: Shape Matching and Object Recognition Using Shape Contexts, IEEE 2002

[21] Chih-Wei Hsu, Chih-Chung Chang, and Chih-Jen Lin: A Practical Guide to Support Vector Classification