Optical Character Recognition

Stelios Barberakis
<chefarov@gmail.com>

Machine learning
Chania, TUC
26/03/2014

Introduction

- What is *Optical Character Recognition (OCP)*?

- Why do we need OCP technology?
  - Usefull applications:
    - Assisting blind and/or visually impaired people
    - Automatic Postcode recognition
    - And more...
  - First small step towards defeating Captcha

Introduction (2)

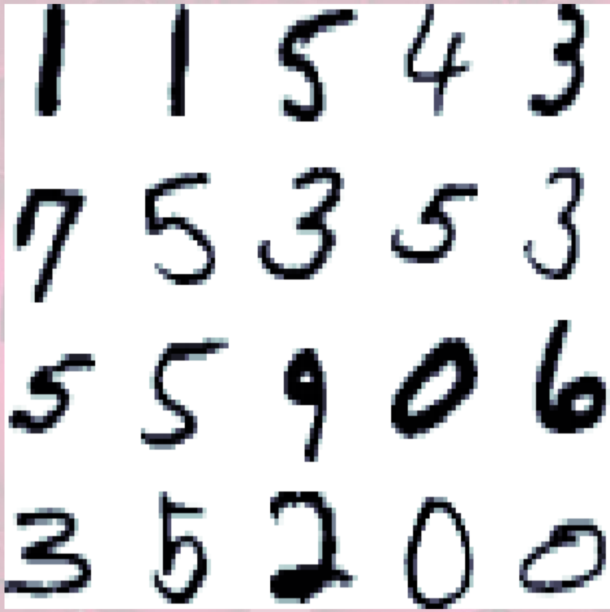
Evaluated some *Machine Learning (ML)* methods/techniques:

- Multilayer Perceptron Networks (MLP)
  - Both a two (2) and three (3) hidden layer
- K-Nearest Neighbors (knn)
- Support Vector Machine (SVM)

Working on the input space

MNIST Dataset

- Used for Comparing methods
- Train set: 60000 sample
- Test set: 10000 samples
- Test set: people not participating in training set
- Resolution: 28x28
- Train set: random ordering
- Test set: 5000 easy, then 5000 harder



Related Work

- MNIST test error rate:
  - 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%

Best implementation so far

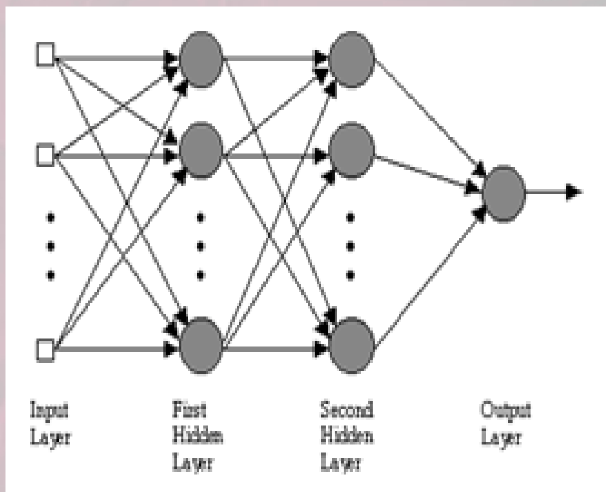
- CNNs: state of the art classifiers for image recognition. Natively translational and scaling invariant
- MCDNN by Ciresan, Meier, Schmidhuber
- Test Error: 0.23%, Human error: 0.2%
- 20/23 of misclassified: correct 2nd guess
- Traffic signs dataset: Achieved half of the human error (1.1% → 0.54%)

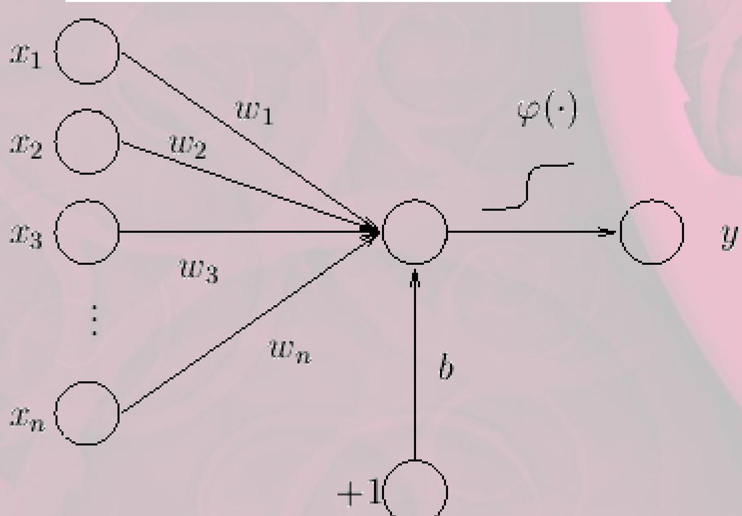
Preprocessing

- Removed padding and scaled pictures to 14x14
- Performance of scaled dataset similar to the one on the original dataset (10% used)
- Scaling intensity values to [-1, 1]. Pictures with different intensity ranges can be classified

note: before calling the trained classifier for a random picture, it must be also normalized to [-1 1]

Multilayer Perceptron Network (MLP)





- Fully connected feed-forward network
- Non-linear classifier
- Input: MxN, M samples, each of N attributes
- Batch (native) or sequential training (faster- preferred for huge datasets).
- 3 layer with sigmoidal activation function can approximate any function to arbitrary accuracy

Backpropagation (BP) – Gradient Descent (GD)

- Learning occurs by adjusting weights in order to decrease error:

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

- Derivation show that in order to follow the gradient downwards we update weights by:

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

- Other BP training functions also exist

Training procedure

Forwards phase:

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

$$h_j = \sum_i x_i v_{ij} \qquad (3.4)$$

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

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

$$h_k = \sum_j a_j w_{jk} \qquad (3.6)$$

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

Backwards phase:

- compute the error at the output using:

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

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

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

- update the output layer weights using:

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

- update the hidden layer weights using:

$$v_{ij} \leftarrow v_{ij} + \eta \delta_{hj} x_i \qquad (3.11)$$

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

Stopping

To avoid overfitting: Earlystopping or L1,L2 regularization (extra term to the loss function)

- EarlyStopping:
  - Iteratively call train(for some epochs). Use validation set to determine how well the network has generalized.
  - Heuristics to decide when to stop.
  - Simplest one: Stop if validation error in last 2 or 3 (earlystopping-iterations) was increasing

Parameters

- Number of layers (2 is usually enough)
- Number of nodes in each layer
- Activation function (each node): sigmoid function: differentiable. Neuron behavior. Gradient descent
- Learning rate(η): How fast it converges. The lower the safer (Reduce oscillations, and avoiding local minimum)
- Momentum(α): Δw depends on the previous Δws. Allows lower (η) and increases convergence speed.

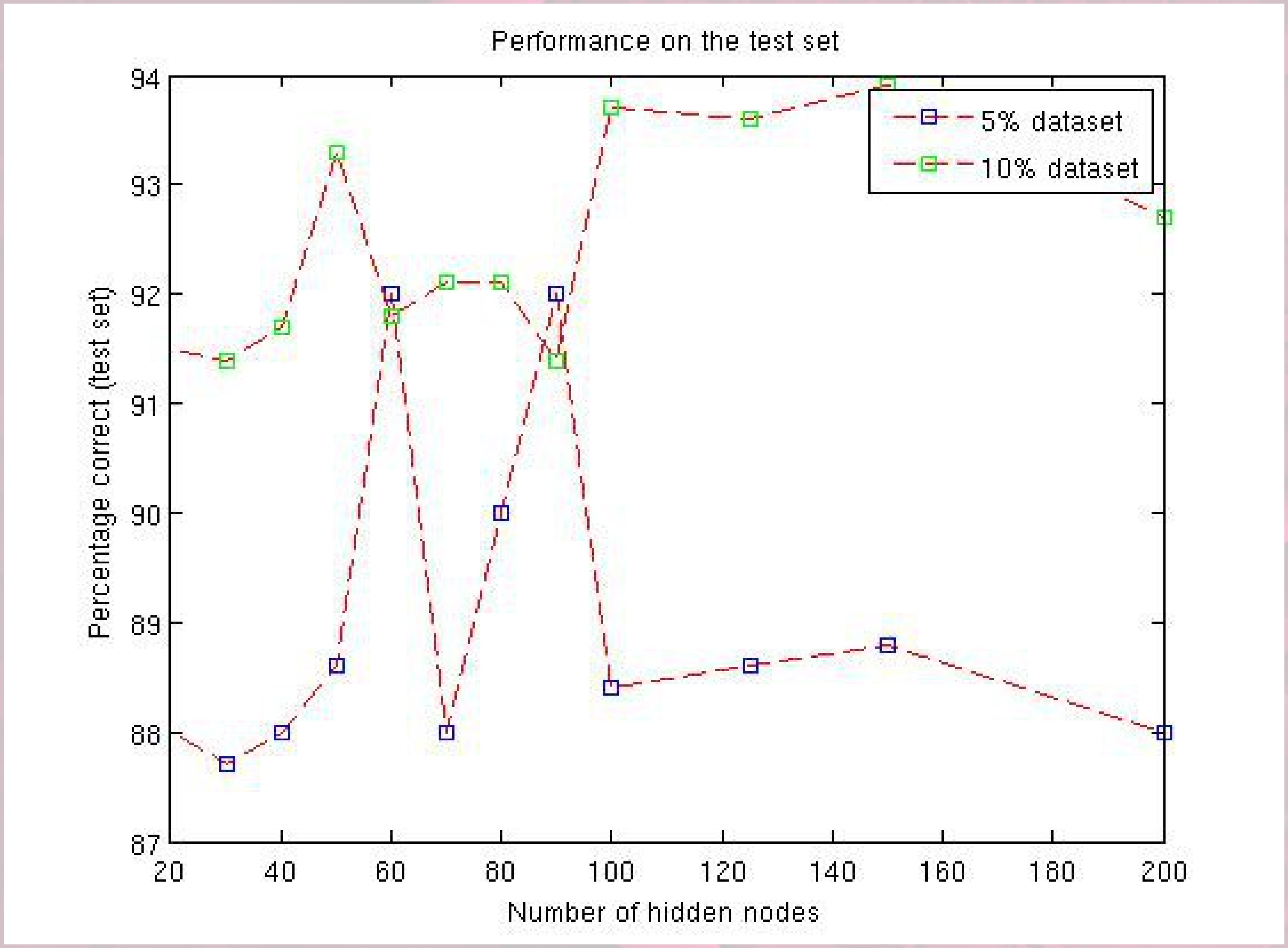
$$w_{ij}^{t} \leftarrow w_{ij}^{t-1} + \eta \delta_o a_j^{\text{hidden}} + \alpha \Delta w_{ij}^{t-1},$$

Applying on MNIST dataset

- Input: Arrays (50000x14x14), reshape to (50000x196)
- Construct MLP (X, T, nodes, layers, outtype)
- Train(X,T, η, n_iterations) #before calling earlystopping!
- Earlystopping(X,T,valid_X,valid_T, η)
- Calculate confusion matrix (test_X, test_T)
- Save trained NN (serialized in disk) using Pickle
- Recall phase: Call forward function for custom input
- Output: 1-of-N encoding using soft-max:

  [0.02, 0.30, 0.14 … 0.05]: classified as '1' (argmax=1)

Performance per architecture (2 layer - MLP)



Performance of MLP

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

- Using Rprop from Pybrain (50 nodes) gave: 99.1% performance !?

Optimize performance

- Parallelize Back propagation: Each thread calculates activation function for each neuron(sequential learning), or dataset is divided in subsets (batch learning).
- Use other training functions instead of Gradient Descent, as is the resilient Back Propagation (Rprop):
  - Each weight has its own update rule in the opposite direction of that weight's partial derivative.
- Best (paranoic) MLP reported:2500-2000-1500-1000-500-10, has performance: 98.43%, affine + elastic preprocessing: 99.69%

Advantages & Disadvantages

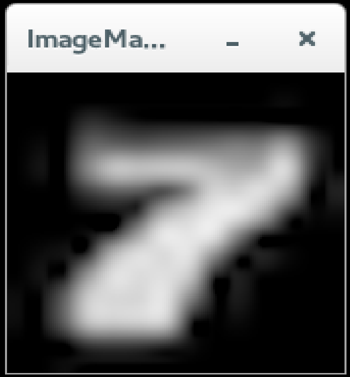
- + Invariant features detection
- + General applicability
- + Can approximate complex models
- - A lot of experimentation needed
- - Training phase too long, need for parallelization
- - Difficult to make use of prior knowledge

Script to classify a character

Parameters:

1. Image path
2. trained net object path
3. Foreground color

```
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
```



kNN

- Non-parametric model, predictions straight from the data. No need for training phase
- For each test point to be classified:
  - ✓ Compute distance (usually euklidean), to every candidate
  - ✓ Predict by averaging the closets k elements.
- Use cross validation to choose k. Not necessary when large datasets are available
- We found out k=3 as the best value for small subsets of the dataset

Performance

- Using the entire dataset we got : 96.4% accuracy (3h 37m), working in input space
- Best performance reported (without preprocessing/FE): 96.91%
- Feature extraction boosts the performance by making the model robust against various input space distortions like (in OCR): translation, rotation, scaling, thinning, etc
- Best results achieved by
  - Shape Contexts (reported 99.37% performance)
  - Deformation models (reported: up to 99.48% performance)

Conclusion on knn

- + 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

SVMs

- Binary classifier. Using multiple binary classifiers (eg: one-versus-one: n(n-1)/2) to perform multiclass classification
- Map the input in higher dimensional space in order to be linearly separable Φ(x)
- Find a linear separating hyperplane with the maximal margin
- Given (x_i,y_i), i=1, . . . , l

  where xi ∈ R^n

  and y ∈ {1, −1}

  optimize:

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

SVMs - steps

- Transform data to the format of an SVM package
- Conduct simple scaling on the data
- Consider the RBF kernel $K(\mathbf{x}, \mathbf{y}) = e^{-\gamma \|\mathbf{x} - \mathbf{y}\|^2}$
- Use cross-validation 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

SVMs – basic kernels

- Polynomial more complex more parameters.
- Linear kernels only when data linearly separable

- 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: $K(\mathbf{x}_i, \mathbf{x}_j) = \tanh(\gamma \mathbf{x}_i{}^T \mathbf{x}_j + r)$.

- using 10% dataset, performed grid search (each parameter 6-7 values), for poly and rbf kernel's parameters:
- RBF kernel: 50min, Polynomial kernel: 2h

performance

Making a grid search on MNIST gave as an optimal

- RBF: (C,γ): C=2.8, γ=0.0073. Performance: 98.6%,. Same performance like the MNIST page reports
- Poly (C,γ,ρ,d): C=0.35, ρ=0.125, γ=0.0625, d=3. Performance: 98,25%, some minutes
- Stronger implementations:
  - Poly: Degree:4 reported 98.9% performance
  - Best: Virtual SVM, deg-9 poly, 2-pixel jittered achieved 99.44%

Conclusion – Future work

- Deeply researched issue – human competitive results
- From individual character recognition to more general tasks
- I don't intent to further study this issue...
- If anyone comes up with this, consider using CNN's