

Face Recognition with OpenCV2

Philipp Wagner
<http://www.bytefish.de>

February 9, 2012

Contents

1	Introduction	1
2	Installation Guide	2
2.1	Installing CMake	2
2.2	Installing MinGW	2
2.3	Building OpenCV	3
3	Generating Data: Random Numbers in OpenCV	4
3.1	Uniform Distribution	4
3.2	Normal Distribution	5
4	Preparing the Training and Test Dataset for OpenCV ML	5
5	Support Vector Machines (SVM)	6
5.1	Definition	6
5.1.1	Non-linear SVM	7
5.2	SVM in OpenCV	7
6	Multi Layer Perceptron	9
6.1	Backpropagation	9
6.2	MLP in OpenCV	9
7	Evaluation	11
7.1	Experiment Data	11
7.2	$y = 2x$	11
7.2.1	Plot	12
7.3	$y = \sin(10x)$	12
7.3.1	Plot	12
7.4	$y = \tan(10x)$	12
7.4.1	Plot	13
A	main.cpp	13

1 Introduction

This document covers the Machine Learning API of the OpenCV2 C++ API. It helps you with setting up your system, gives a brief introduction into Support Vector Machines and Neural Networks and shows how it's implemented with OpenCV. Machine Learning is a branch of Artificial Intelligence and concerned with the question how to make machines able to learn from data. The core idea is to enable a machine to make intelligent decisions and predictions, based on experiences from the past. Algorithms of Machine Learning require interdisciplinary knowledge and often intersect with topics of statistics, mathematics, physics, pattern recognition and more.

OpenCV2 comes with a machine learning library for:

- Decision Trees
- Boosting
- Support Vector Machines
- Expectation Maximization
- Neural Networks

[OpenCV \(Open Source Computer Vision\)](#) is a popular computer vision library started by [Intel](#) in 1999. The cross-platform library sets its focus on real-time image processing and includes patent-free implementations of the latest computer vision algorithms. In 2008 [Willow Garage](#) took over support and OpenCV 2.3.1 now comes with a programming interface to C, C++, [Python](#) and [Android](#). OpenCV is released under a BSD license, so it is used in academic and commercial projects such as [Google Streetview](#).

Please don't copy and paste the code from this document, the project has been uploaded to <http://www.github.com/bytefish/opencv>. All code is released under a [BSD license](#), so feel free to use it for your projects.

2 Installation Guide

This installation guide explains how to install the software for this document. [CMake](#) is used as build system for the examples, [MinGW \(Minimalist GNU for Windows\)](#) is used as the compiler for Windows and OpenCV2 is compiled from source. There are binaries for OpenCV2 already, so why is it useful to build it from source at all? Your architecture may not be supported by the binaries, your toolchain may differ or the OpenCV version in your repository may not be the latest. Please note: You can always use the binaries supplied by WillowGarage or the binaries supplied by your distribution if they work for you.

The following guide was tested on Microsoft Windows XP SP3 and Ubuntu 10.10.

2.1 Installing CMake

[CMake](#) is an open-source, cross-platform build system. It manages the build process in a compiler-independent manner and is able to generate native build environments to compile the source code ([Make](#), [Apple Xcode](#), [Microsoft Visual Studio](#), [MinGW](#), ...). Projects like [OpenCV](#), [KDE](#) or [Blender 3D](#) recently switched to CMake due to its flexibility. The CMake build process itself is controlled by configuration files, placed in the source directory (called `CMakeLists.txt`). Each `CMakeLists.txt` consists of CMake commands in the form of `COMMAND(arguments...)`, that describe how to include header files, build libraries and executables. Please see the [CMake Documentation](#) for a list of available commands. A Windows installer is available at cmake.org/resources/software.html (called `cmake-<version>-win32-x86.exe`). Make sure to select *"Add CMake to the system PATH for all users"* during setup or manually add it, so you can use `cmake`, `ccmake` and the `cmake-gui` from command line (see [Microsoft Support: How To Manage Environment Variables in Windows XP](#) for details). Linux users should check the repository of their distribution, because the CMake binaries are often available already. If CMake is not available one can build it from source by:

```
./bootstrap
make
make install
```

Or install generic Linux binaries (called `cmake-<version>-<os>-<architecture>.sh`):

```
sudo sh cmake-<version>-<os>-<architecture>.sh --prefix=/usr/local
```

2.2 Installing MinGW

[MinGW \(Minimalist GNU for Windows\)](#) is a port of the [GNU Compiler Collection \(GCC\)](#) and can be used for the development of native [Microsoft Windows](#) applications. The easiest way to install MinGW is to use the automated mingw-get-installer from [sourceforge.net/projects/mingw/files/Automated MinGW Installer/mingw-get-inst/](http://sourceforge.net/projects/mingw/files/Automated%20MinGW%20Installer/mingw-get-inst/) (called `mingw-get-inst-20101030.exe` at time of writing this). If the path to the download changes, please navigate there from mingw.org.

Make sure to select *"C++ Compiler"* in the *Compiler Suite* dialog during setup. Since MinGW doesn't add its binaries to the Windows PATH environment, you'll need to manually add it. The MinGW Page says: *Add c:\MinGW\bin to the PATH environment variable by opening the System control panel, going to the Advanced tab, and clicking the Environment Variables button. If you currently have a Command Prompt window open, it will not recognize the change to the environment variables; you will need to open a new Command Prompt window to get the new PATH.*

Linux users should install [gcc](#) and [make](#) (or a build tool supported by CMake) from the repository of their distribution. In Ubuntu the `build-essential` package contains all necessary tools to get started, in Fedora and SUSE you'll need to install it from the available development tools.

2.3 Building OpenCV

Please skip this section if you are using the OpenCV binaries supplied by WillowGarage or your distribution. To build OpenCV you'll need CMake (see section 2.1), a C/C++ compiler (see section 2.2) and the OpenCV source code. At time of writing this, the latest OpenCV sources are available at <http://sourceforge.net/projects/opencvlibrary/>. I've heard the OpenCV page will see some changes soon, so if the sourceforge isn't used for future versions anymore navigate from the official page: <http://opencv.willowgarage.com>.

In this guide I'll use [OpenCV 2.3.0](#) for Windows and [OpenCV 2.3.1](#) for Linux. If you need the latest Windows version download the [superpack](#), which includes binaries and sources for Windows.

Create the build folder

First of all extract the source code to a folder of your choice, then open a terminal and `cd` into this folder. Then create a folder `build`, where we will build OpenCV in:

```
mkdir build
cd build
```

Build OpenCV in Windows

Now we'll create the Makefiles to build OpenCV. You need to specify the path you want to install OpenCV to (e.g. `c:/opencv`), preferably it's not the source folder. Note, that CMake expects a slash (`/`) as path separator. So if you are using Windows you'll now write:

```
cmake -G "MinGW Makefiles" -D:CMAKE_BUILD_TYPE=RELEASE -D:BUILD_EXAMPLES=1 -D:
CMAKE_INSTALL_PREFIX=C:/opencv ..
mingw32-make
mingw32-make install
```

Usually CMake is good at guessing the parameters, but there are a lot more options you can set (for Qt, Python, ..., see [WillowGarage's Install Guide](#) for details). It's a good idea to use the `cmake-gui` to see and set the available switches. For now you can stick to the Listing, it works fine for Windows and Linux.

Better get a coffee, because OpenCV takes a while to compile! Once it is finished and you've decided to build dynamic libraries (assumed in this installation guide), you have to add the `bin` path of the installation to Windows PATH variable (e.g. `c:/opencv/bin`). If you don't know how to do that, see [Microsoft Support: How To Manage Environment Variables in Windows XP](#) for details.

Build OpenCV in Linux

Creating the Makefiles in Linux is (almost) similar to Windows. Again choose a path you want to install OpenCV to (e.g. `/usr/local`), preferably it's not the source folder.

```
1 cmake -D CMAKE_BUILD_TYPE=RELEASE -D BUILD_EXAMPLES=1 -D CMAKE_INSTALL_PREFIX=/usr/
   local ..
2 make
3 sudo make install
```

Sample CMakeLists.txt

Once CMake is installed a simple `CMakeLists.txt` is sufficient for building an OpenCV project:

```
# set the minimum cmake version
CMAKE_MINIMUM_REQUIRED(VERSION 2.8)
# project name
PROJECT(hello_opencv)
# you probably need to set this
SET(OpenCV_DIR /path/to/your/opencv/installation)
# finds OpenCV
FIND_PACKAGE(OpenCV REQUIRED)
# build the executable from main.cpp
ADD_EXECUTABLE(hellocv main.cpp)
# link against the opencv libraries
TARGET_LINK_LIBRARIES(hellocv ${OpenCV_LIBS})
```

To build the project one would simply do (assuming we're in the folder with `CMakeLists.txt`):

```
# create build directory
mkdir build
# ... and cd into
cd build
# generate platform-dependent makefiles
cmake ..
# build the project
make
# run the executable
./hellocv
```

Or if you are on Windows with MinGW you would do:

```
mkdir build
cd build
cmake -G "MinGW Makefiles" ..
mingw32-make
```

3 Generating Data: Random Numbers in OpenCV

We want to see how the Machine Learning algorithms perform on linearly separable and non-linearly separable datasets. For this we'll simply generate some random data from several functions. Each thread in OpenCV has access to a default random number generator `cv::theRNG()`. For a single pass the seed for the random number generator doesn't have to be set, but if many random numbers have to be generated (for example in a loop) the seed has to be set.

This can be done by assigning the local time to the random number generator:

```
cv::theRNG() = cv::RNG(time(0))
```

3.1 Uniform Distribution

Uniform Distributions can be generated in OpenCV with the help of `cv::randu`. The signature of `cv::randu` is:

```
void randu(Mat& mtx, const Scalar& low, const Scalar& high);
```

Where

- **mtx** is the Matrix to be filled with uniform distributed random numbers
- **low** is the inclusive lower boundary of generated random numbers
- **high** is the exclusive upper boundary of generated random numbers

3.2 Normal Distribution

A Normal Distribution can be generated in OpenCv with the help of *cv::randn*.

```
void randn(Mat& mtx, const Scalar& mean, const Scalar& stddev);
```

Where

- **mtx** is the Matrix to be filled with normal distributed random numbers
- **mean** the mean value of the generated random numbers
- **stddev** the standard deviation of the generated random numbers

4 Preparing the Training and Test Dataset for OpenCV ML

In the C++ Machine Learning API of OpenCV training and test data is given as a *cv::Mat* matrix. The constructor of *cv::Mat* is defined as:

```
Mat::Mat(int rows, int cols, int type);
```

Where

- rows is the number of samples (for all classes!)
- columns is the number of dimensions
- type is the image type

In the machine learning library of OpenCV each row or column in the training data is a n-dimensional sample. The default ordering is row sampling and class labels are given in a matrix with equal length (one column only, of course).

```
cv::Mat trainingData(numTrainingPoints, 2, CV_32FC1);
cv::Mat testData(numTestPoints, 2, CV_32FC1);

cv::randu(trainingData, 0, 1);
cv::randu(testData, 0, 1);

cv::Mat trainingClasses = labelData(trainingData, equation);
cv::Mat testClasses = labelData(testData, equation);
```

Since only binary classification problems are considered the function *f* returns the classes -1 and 1 for a given two-dimensional data point:

```
// function to learn
int f(float x, float y, int equation) {
    switch(equation) {
        case 0:
            return y > sin(x*10) ? -1 : 1;
            break;
        case 1:
            return y > cos(x * 10) ? -1 : 1;
            break;
        case 2:
            return y > 2*x ? -1 : 1;
            break;
        case 3:
            return y > tan(x*10) ? -1 : 1;
            break;
        default:
            return y > cos(x*10) ? -1 : 1;
    }
}
```

And to label data one can use the function *labelData*:

```
cv::Mat labelData(cv::Mat points, int equation) {
    cv::Mat labels(points.rows, 1, CV_32FC1);
    for(int i = 0; i < points.rows; i++) {
        float x = points.at<float>(i,0);
        float y = points.at<float>(i,1);
```

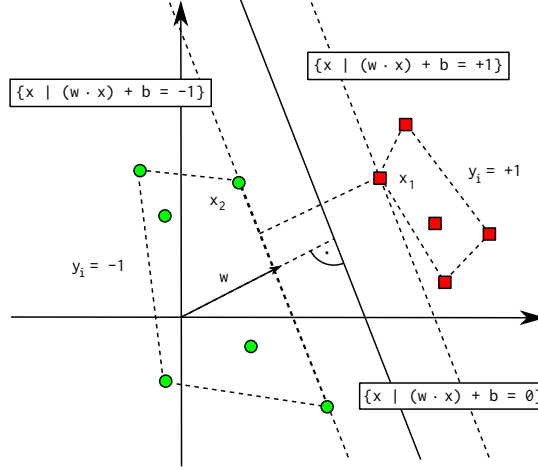


Figure 1: Maximum Margin Classifier.

```

    labels.at<float>(i, 0) = f(x, y, equation);
}
return labels;
}

```

5 Support Vector Machines (SVM)

Support Vector Machines were first introduced by Vapnik and Chervonenkis in [VC74]. The core idea is to find the optimal hyperplane to separate a dataset, while there are theoretically infinite hyperplanes to separate the dataset. A hyperplane is chosen, so that the distance to the nearest datapoint of both classes is maximized (Figure 1). The points spanning the hyperplane are the *Support Vectors*, hence the name *Support Vector Machines*. [CV95]

5.1 Definition

Given a Set of Datapoints \mathcal{D} :

$$\mathcal{D} = \{(x_i, y_i) | x_i \in \mathbb{R}^p, y_i \in \{-1, 1\}\}_{i=1}^n$$

where

- x_i is a point in p-dimensional vector
- y_i is the corresponding class label

We search for $\omega \in \mathbb{R}^n$ and bias b , forming the Hyperplane H:

$$\omega^T x + b = 0$$

that separates both classes so that:

$$\begin{aligned} \omega^T x + b &= 1, \text{ if } y = 1 \\ \omega^T x + b &= -1, \text{ if } y = -1 \end{aligned}$$

The optimization problem that needs to be solved is:

$$\min \frac{1}{2} \omega^T \omega$$

subject to:

$$\begin{aligned} \omega^T x + b &\geq 1, y = 1 \\ \omega^T x + b &\leq -1, y = -1 \end{aligned}$$

Such quadratic optimization problems can be solved with standard solvers, such as [GNU Octave](#) or [Matlab](#).

5.1.1 Non-linear SVM

The kernel trick is used for classifying non-linear datasets. It works by transforming data points into a higher dimensional feature space with a *kernel function*, where the dataset can be separated again (see Figure 2).

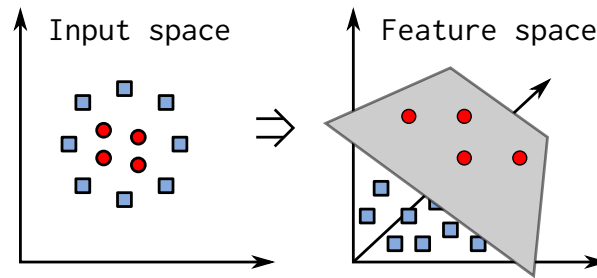


Figure 2: Kernel Trick

Commonly used kernel functions are *RBF kernels*:

$$k(x, x') = \exp\left(-\frac{\|x - x'\|^2}{\sigma^2}\right)$$

or *polynomial kernels*:

$$k(x, x') = (x \cdot x')^d$$

.

5.2 SVM in OpenCV

Parameters for a SVM have to be defined in the structure *CvSVMParams*.

Parameters

Listing 1: Example CvSVMParams

```
CvSVMParams param = CvSVMParams();

param.svm_type = CvSVM::C_SVC;
param.kernel_type = CvSVM::LINEAR;

param.degree = 0; // for poly
param.gamma = 20; // for poly/rbf/sigmoid
param.coef0 = 0; // for poly/sigmoid

param.C = 7; // for CV_SVM_C_SVC, CV_SVM_EPS_SVR and CV_SVM_NU_SVR
param.nu = 0.0; // for CV_SVM_NU_SVC, CV_SVM_ONE_CLASS, and CV_SVM_NU_SVR
param.p = 0.0; // for CV_SVM_EPS_SVR

param.class_weights = NULL; // for CV_SVM_C_SVC
param.term_crit.type = CV_TERMCRIT_ITER | CV_TERMCRIT_EPS;
param.term_crit.max_iter = 1000;
param.term_crit.epsilon = 1e-6;
```

Where the parameters are (taken from the OpenCV documentation):

- **svm_type**
 - `CvSVM::C_SVC` n-class classification ($n \geq 2$), allows imperfect separation of classes with penalty multiplier C for outliers.
 - `CvSVM::NU_SVC` n-class classification with possible imperfect separation. Parameter ν (in the range $0 \dots 1$, the larger the value, the smoother the decision boundary) is used instead of C .

- `CvSVM::ONE_CLASS` one-class SVM. All the training data are from the same class, SVM builds a boundary that separates the class from the rest of the feature space.
 - `CvSVM::EPS_SVR` regression. The distance between feature vectors from the training set and the fitting hyper-plane must be less than p . For outliers the penalty multiplier C is used.
 - `CvSVM::NU_SVR` regression; ν is used instead of p .
- `kernel_type`
 - `CvSVM::LINEAR` no mapping is done, linear discrimination (or regression) is done in the original feature space. It is the fastest option. $d(x, y) = x \cdot y == (x, y)$.
 - `CvSVM::POLY` polynomial kernel: $d(x, y) = (gamma * (x \cdot y) + coef0)^{degree}$.
 - `CvSVM::RBF` radial-basis-function kernel; a good choice in most cases: $d(x, y) = exp(-gamma * |x - y|^2)$.
 - `CvSVM::SIGMOID` sigmoid function is used as a kernel: $d(x, y) = tanh(gamma * (x \cdot y) + coef0)$.
 - `C`, `nu`, `p` Parameters in the generalized SVM optimization problem.
 - `class_weights` Optional weights, assigned to particular classes. They are multiplied by C and thus affect the misclassification penalty for different classes. The larger weight, the larger penalty on misclassification of data from the corresponding class.
 - `term_criteria` Termination procedure for iterative SVM training procedure (which solves a partial case of constrained quadratic optimization problem)
 - `type` is either `CV_TERMCRIT_ITER` or `CV_TERMCRIT_ITER`
 - `max_iter` is the maximum number of iterations in training.
 - `epsilon` is the error to stop training.

Training

Training can either be done by passing the vector with the training data and vector with the corresponding class labels to the constructor or the train method.

```
CvSVM(const CvMat* _train_data,
      const CvMat* _responses,
      const CvMat* _var_idx=0,
      const CvMat* _sample_idx=0,
      CvSVMParams _params=CvSVMParams());
```

where

- `_train_data` is a Matrix with the n -dimensional feature vectors
- `_responses` is a vector with the class for the corresponding feature vector
- `_var_idx` identifies features of interest (can be left empty for this example, in code: `cv::Mat()`)
- `_sample_idx` identifies samples of interest (can be left empty for this example, in code: `cv::Mat()`)
- `_params` Parameter for the SVM from Listing 1

This applies to the train method aswell:

```
virtual bool train(const CvMat* _train_data,
                  const CvMat* _responses,
                  const CvMat* _var_idx=0,
                  const CvMat* _sample_idx=0,
                  CvSVMParams _params=CvSVMParams());
```

The `train` methods of the SVM has some limitations (at time of writing this):

- Only `CV_ROW_SAMPLE` is supported
- Missing measurements are not supported

The `train_auto` method finds the best parameters with a Gridsearch and a k -fold cross validation. This method is available for OpenCV Versions ≥ 2.0 .

Prediction

Self explaining code.

```
for(int i = 0; i < testData.rows; i++) {
    cv::Mat sample = testData.row(i);
    float result = svm.predict(sample);
}
```

Support Vectors

The support vectors of a SVM can be obtained using the `get_support_vector` function of the API:

```
int svec_count = svm.get_support_vector_count();
for(int vecNum = 0; vecNum < svec_count; vecNum++) {
    const float* vec = svm.get_support_vector(vecNum);
}
```

A complete example for Support Vector Machines in OpenCV is given in the Appendix.

6 Multi Layer Perceptron

An Artificial Neural Network is a biological inspired computational model. *Inputs* multiplied by *weights* result in an *activation* and form the *output* of a network. Research in Artificial Neural Networks (ANN) began in 1943, when McCulloch and Pitts gave a definition of a formal neuron in their paper "A Logical Calculus of the Ideas Immanent in Nervous Activity" [MP43]. In 1958 Rosenblatt invented the perceptron, which is a simple feedforward neural network. The downfall of the perceptron algorithm is that it only converges on lineary seperable datasets and is not able to solve non-linear problems such as the XOR problem. This was proven by Minsky and Papert in their monograph "Perceptrons", but they showed that a two-layer feedforward architecture can overcome this limitation. It was until 1986 when Rumelhart, Hinton and Williams presented a learning rule for Arificial Neural Networks with hidden units in their paper "Learning Internal Representations by Error Propagation". The original discovery of backpropagation is actually credited to Werbos who described the algorithm in his 1974 Ph.D. thesis at Havard University, see [Wer94] for the roots of backpropagation.

A detailed introduction to Pattern Recognition with Neural Networks is given by [Bis95].

6.1 Backpropagation

1. Initilaize weights with random values
2. Present the input vector to the network
3. Evaluate the output of the network after a forward propagation of the signal
4. Calculate $\delta_j = (y_j - d_j)$ where d_j is the target output of neuron j and y_j is the actual output $y_j = g(\sum_i w_{ij}x_i) = (1 + e^{-\sum_i w_{ij}x_i})^{-1}$, (when the activation function is of a sigmoid type).
5. For all other neurons (from the first to the last layer) calculate $\delta_j = \sum_k w_{jk}g'(x)\delta_k$, where δ_k is δ_j of the succeeding layer and $g'(x) = y_k(1 - y_k)$
6. Update weights with $w_{ij}(t+1) = w_{ij}(t) - \eta y_i y_j (1 - y_j) \delta_j$, where η is the learning rate.
7. Termination Criteria. Goto Step 2 for a fixed number of iterations or an error.

The network error is defined as:

$$E = \frac{1}{2} \sum_{j=1}^m (d_j - y_j)^2$$

6.2 MLP in OpenCV

A Multilayer Perceptron in OpenCV is an instance of `CvANN_MLP`.

```
CvANN_MLP mlp;
```

Parameters

The performance of a Multilayer perceptron depends on its parameters:

```
CvTermCriteria criteria;  
criteria.max_iter = 100;  
criteria.epsilon = 0.00001f;  
criteria.type = CV_TERMCRIT_ITER | CV_TERMCRIT_EPS;  
  
CvANN_MLP_TrainParams params;  
params.train_method = CvANN_MLP_TrainParams::BACKPROP;  
params.bp_dw_scale = 0.05f;  
params.bp_moment_scale = 0.05f;  
params.term_crit = criteria;
```

Where the parameters are (taken from the OpenCV 1.0 documentation¹):

- **term_crit** The termination criteria for the training algorithm. It identifies how many iterations is done by the algorithm (for sequential backpropagation algorithm the number is multiplied by the size of the training set) and how much the weights could change between the iterations to make the algorithm continue.
- **train_method** The training algorithm to use; can be one of `CvANN_MLP_TrainParams::BACKPROP` (sequential backpropagation algorithm) or `CvANN_MLP_TrainParams::RPROP` (RPROP algorithm, default value).
- **bp_dw_scale** (Backpropagation only): The coefficient to multiply the computed weight gradient by. The recommended value is about 0.1.
- **bp_moment_scale** (Backpropagation only): The coefficient to multiply the difference between weights on the 2 previous iterations. This parameter provides some inertia to smooth the random fluctuations of the weights. It can vary from 0 (the feature is disabled) to 1 and beyond. The value 0.1 or so is good enough.
- **rp_dw0** (RPROP only): Initial magnitude of the weight delta. The default value is 0.1.
- **rp_dw_plus** (RPROP only): The increase factor for the weight delta. It must be > 1 , default value is 1.2 that should work well in most cases, according to the algorithm's author.
- **rp_dw_minus** (RPROP only): The decrease factor for the weight delta. It must be < 1 , default value is 0.5 that should work well in most cases, according to the algorithm's author.
- **rp_dw_min** (RPROP only): The minimum value of the weight delta. It must be > 0 , the default value is `FLT_EPSILON`.
- **rp_dw_max** (RPROP only): The maximum value of the weight delta. It must be > 1 , the default value is 50.

Layers

The purpose of a neural network is to generalize, which is the ability to approximate outputs for inputs not available in the training set. [Sar02] While small networks may not be able to approximate a function, large networks tend to overfit and not find any relationship in data.² It has been shown that, given enough data, a multi layer perceptron with one hidden layer can approximate any continuous function to any degree of accuracy. [HSW92]

The number of neurons per layer is stored in a row-ordered `cv::Mat`.

```
cv::Mat layers = cv::Mat(4, 1, CV_32SC1);  
  
layers.row(0) = cv::Scalar(2);  
layers.row(1) = cv::Scalar(10);  
layers.row(2) = cv::Scalar(15);  
layers.row(3) = cv::Scalar(1);  
  
mlp.create(layers);
```

¹http://www.cognitics.com/opencv/docs/1.0/ref/opencvref_ml.htm

²The model describes random error or noise instead of the relationship of the data.

Training

The API for training a multilayer perceptron takes the training data, training classes and the structure for the parameters.

```
mlp.train(trainingData, trainingClasses, cv::Mat(), cv::Mat(), params);
```

Prediction

The API for the prediction is slightly different from the SVM API. Activations of the output layer are stored in a `cv::Mat` response, simply because one can design neural networks with multiple neurons in the output layer.

Since the problem used in this example is a binary classification problem, it is sufficient to have only one neuron in the output layer. It is therefore the only activation to check.

```
mlp.predict(sample, response);  
float result = response.at<float>(0,0);
```

7 Evaluation

In this section the following algorithms will be used for classification:

- Support Vector Machine
- Multi Layer Perceptron
- k-Nearest-Neighbor
- Normal Bayes
- Decision Tree

To evaluate a predictor it is possible to calculate its accuracy. For two classes it is given as:

$$Accuracy = \frac{\text{true positive}}{\text{true positive} + \text{false positive}}$$

The performance of Support Vector Machines and especially Neural Networks depend on the parameters chosen. In case of a neural network it is difficult to find the appropriate parameters and architecture. Designing an Artificial Neural Network is often more a rule of thumb and networks should be optimized iteratively starting with one hidden layer and few neurons. Parameters for a Support Vector Machine can be estimated using Cross Validation and Grid Search (both can be used as `train_auto` in OpenCV ≥ 2.0).

Parameters are not optimized in this experiment, remember to optimize the parameters yourself when using one of the algorithms.

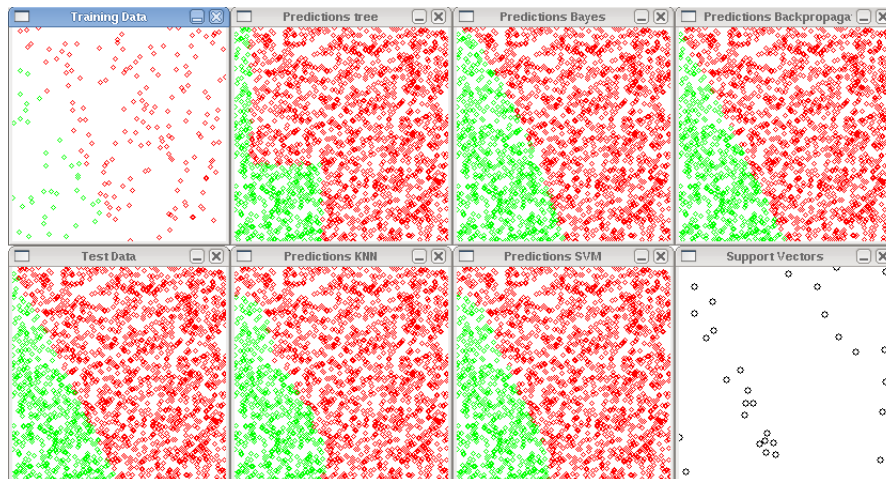
7.1 Experiment Data

In this experiment linear and non-linear functions are learned. 200 points for training and 2000 points for testing are generated.

7.2 $y = 2x$

Predictor	Accuracy
Support Vector Machine	0.99
Multi Layer Perceptron (2, 10, 15, 1)	0.994
k-Nearest-Neighbor (k = 3)	0.9825
Normal Bayes	0.9425
Decision Tree	0.923

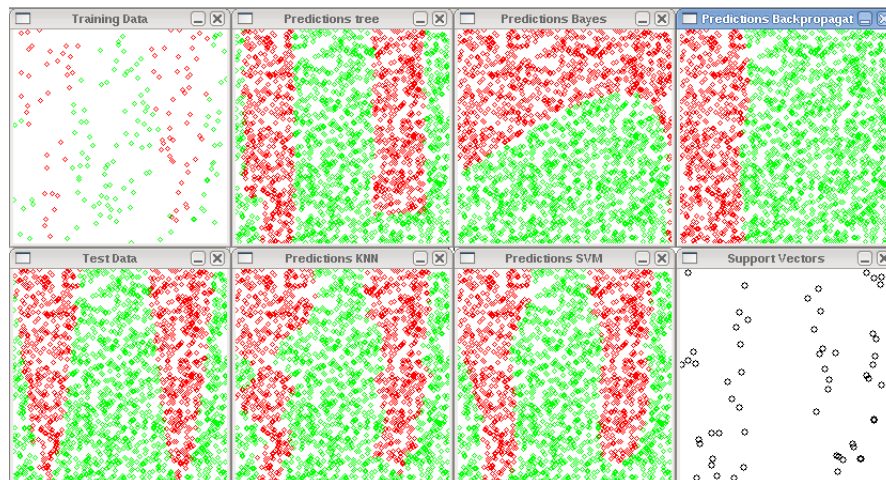
7.2.1 Plot



7.3 $y = \sin(10x)$

Predictor	Accuracy
Support Vector Machine	0.913
Multi Layer Perceptron (2, 10, 15, 1)	0.6855
k-Nearest-Neighbor (k = 3)	0.9
Normal Bayes	0.632
Decision Tree	0.886

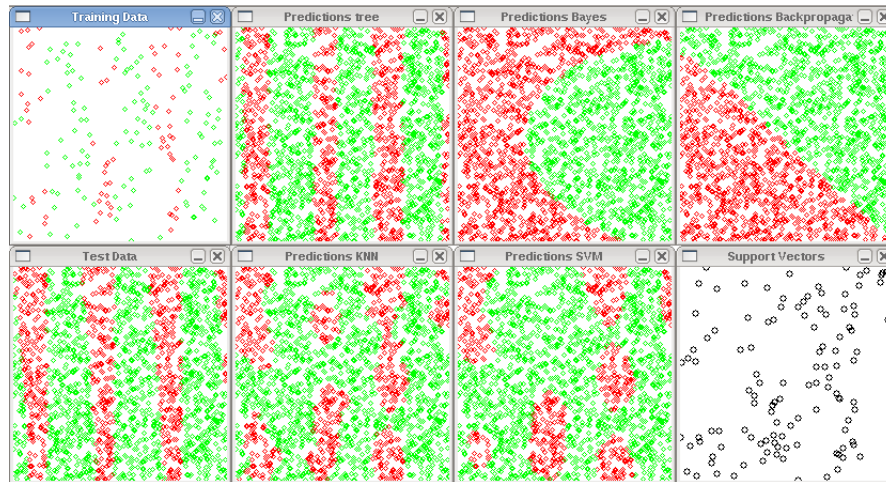
7.3.1 Plot



7.4 $y = \tan(10x)$

Predictor	Accuracy
Support Vector Machine	0.7815
Multi Layer Perceptron (2, 10, 15, 1)	0.5115
k-Nearest-Neighbor (k = 3)	0.8195
Normal Bayes	0.542
Decision Tree	0.9155

7.4.1 Plot



A main.cpp

Listing 2: main.cpp

```
#include <iostream>
#include <math.h>
#include <string>
#include "cv.h"
#include "ml.h"
#include "highgui.h"

using namespace cv;
using namespace std;

bool plotSupportVectors=true;
int numTrainingPoints=200;
int numTestPoints=2000;
int size=200;
int eq=0;

// accuracy
float evaluate(cv::Mat& predicted, cv::Mat& actual) {
    assert(predicted.rows == actual.rows);
    int t = 0;
    int f = 0;
    for(int i = 0; i < actual.rows; i++) {
        float p = predicted.at<float>(i,0);
        float a = actual.at<float>(i,0);
        if((p >= 0.0 && a >= 0.0) || (p <= 0.0 && a <= 0.0)) {
            t++;
        } else {
            f++;
        }
    }
    return (t * 1.0) / (t + f);
}

// plot data and class
void plot_binary(cv::Mat& data, cv::Mat& classes, string name) {
    cv::Mat plot(size, size, CV_8UC3);
    plot.setTo(cv::Scalar(255.0,255.0,255.0));
    for(int i = 0; i < data.rows; i++) {
        float x = data.at<float>(i,0) * size;
        float y = data.at<float>(i,1) * size;

        if(classes.at<float>(i, 0) > 0) {
            cv::circle(plot, Point(x,y), 2, CV_RGB(255,0,0),1);
        }
    }
}
```

```

    } else {
        cv::circle(plot, Point(x,y), 2, CV_RGB(0,255,0),1);
    }
}
cv::imshow(name, plot);
}

// function to learn
int f(float x, float y, int equation) {
    switch(equation) {
        case 0:
            return y > sin(x*10) ? -1 : 1;
            break;
        case 1:
            return y > cos(x * 10) ? -1 : 1;
            break;
        case 2:
            return y > 2*x ? -1 : 1;
            break;
        case 3:
            return y > tan(x*10) ? -1 : 1;
            break;
        default:
            return y > cos(x*10) ? -1 : 1;
    }
}

// label data with equation
cv::Mat labelData(cv::Mat points, int equation) {
    cv::Mat labels(points.rows, 1, CV_32FC1);
    for(int i = 0; i < points.rows; i++) {
        float x = points.at<float>(i,0);
        float y = points.at<float>(i,1);
        labels.at<float>(i, 0) = f(x, y, equation);
    }
    return labels;
}

void svm(cv::Mat& trainingData, cv::Mat& trainingClasses, cv::Mat& testData, cv::Mat&
testClasses) {
    CvSVMParams param = CvSVMParams();

    param.svm_type = CvSVM::C_SVC;
    param.kernel_type = CvSVM::RBF; //CvSVM::RBF, CvSVM::LINEAR ...
    param.degree = 0; // for poly
    param.gamma = 20; // for poly/rbf/sigmoid
    param.coef0 = 0; // for poly/sigmoid

    param.C = 7; // for CV_SVM_C_SVC, CV_SVM_EPS_SVR and CV_SVM_NU_SVR
    param.nu = 0.0; // for CV_SVM_NU_SVC, CV_SVM_ONE_CLASS, and CV_SVM_NU_SVR
    param.p = 0.0; // for CV_SVM_EPS_SVR

    param.class_weights = NULL; // for CV_SVM_C_SVC
    param.term_crit.type = CV_TERMCRIT_ITER +CV_TERMCRIT_EPS;
    param.term_crit.max_iter = 1000;
    param.term_crit.epsilon = 1e-6;

    // SVM training (use train auto for OpenCV>=2.0)
    CvSVM svm(trainingData, trainingClasses, cv::Mat(), cv::Mat(), param);

    cv::Mat predicted(testClasses.rows, 1, CV_32F);

    for(int i = 0; i < testData.rows; i++) {
        cv::Mat sample = testData.row(i);

        float x = sample.at<float>(0,0);
        float y = sample.at<float>(0,1);

        predicted.at<float>(i, 0) = svm.predict(sample);
    }

    cout << "Accuracy_{SVM} = " << evaluate(predicted, testClasses) << endl;
}

```



```

plot_binary(testData, predicted, "Predictions SVM");

// plot support vectors
if(plotSupportVectors) {
    cv::Mat plot_sv(size, size, CV_8UC3);
    plot_sv.setTo(cv::Scalar(255.0,255.0,255.0));

    int svec_count = svm.get_support_vector_count();
    for(int vecNum = 0; vecNum < svec_count; vecNum++) {
        const float* vec = svm.get_support_vector(vecNum);
        cv::circle(plot_sv, Point(vec[0]*size, vec[1]*size), 3 , CV_RGB(0, 0, 0));
    }
    cv::imshow("Support Vectors", plot_sv);
}
}

void mlp(cv::Mat& trainingData, cv::Mat& trainingClasses, cv::Mat& testData, cv::Mat&
testClasses) {

    cv::Mat layers = cv::Mat(4, 1, CV_32SC1);

    layers.row(0) = cv::Scalar(2);
    layers.row(1) = cv::Scalar(10);
    layers.row(2) = cv::Scalar(15);
    layers.row(3) = cv::Scalar(1);

    CvANN_MLP mlp;
    CvANN_MLP_TrainParams params;
    CvTermCriteria criteria;
    criteria.max_iter = 100;
    criteria.epsilon = 0.00001f;
    criteria.type = CV_TERMCRIT_ITER | CV_TERMCRIT_EPS;
    params.train_method = CvANN_MLP_TrainParams::BACKPROP;
    params.bp_dw_scale = 0.05f;
    params.bp_moment_scale = 0.05f;
    params.term_crit = criteria;

    mlp.create(layers);

    // train
    mlp.train(trainingData, trainingClasses, cv::Mat(), cv::Mat(), params);

    cv::Mat response(1, 1, CV_32FC1);
    cv::Mat predicted(testClasses.rows, 1, CV_32F);
    for(int i = 0; i < testData.rows; i++) {
        cv::Mat response(1, 1, CV_32FC1);
        cv::Mat sample = testData.row(i);

        mlp.predict(sample, response);
        predicted.at<float>(i,0) = response.at<float>(0,0);
    }

    cout << "Accuracy_{MLP} = " << evaluate(predicted, testClasses) << endl;
    plot_binary(testData, predicted, "Predictions Backpropagation");
}

void knn(cv::Mat& trainingData, cv::Mat& trainingClasses, cv::Mat& testData, cv::Mat&
testClasses, int K) {

    CvKNearest knn(trainingData, trainingClasses, cv::Mat(), false, K);
    cv::Mat predicted(testClasses.rows, 1, CV_32F);
    for(int i = 0; i < testData.rows; i++) {
        const cv::Mat sample = testData.row(i);
        predicted.at<float>(i,0) = knn.find_nearest(sample, K);
    }

    cout << "Accuracy_{KNN} = " << evaluate(predicted, testClasses) << endl;
    plot_binary(testData, predicted, "Predictions KNN");
}
}

```

```

void bayes(cv::Mat& trainingData, cv::Mat& trainingClasses, cv::Mat& testData, cv::Mat
& testClasses) {

    CvNormalBayesClassifier bayes(trainingData, trainingClasses);
    cv::Mat predicted(testClasses.rows, 1, CV_32F);
    for (int i = 0; i < testData.rows; i++) {
        const cv::Mat sample = testData.row(i);
        predicted.at<float>(i, 0) = bayes.predict(sample);
    }

    cout << "Accuracy_{BAYES} = " << evaluate(predicted, testClasses) << endl;
    plot_binary(testData, predicted, "Predictions Bayes");
}

void decisiontree(cv::Mat& trainingData, cv::Mat& trainingClasses, cv::Mat& testData,
cv::Mat& testClasses) {

    CvDTree dtree;
    cv::Mat var_type(3, 1, CV_8U);

    // define attributes as numerical
    var_type.at<unsigned int>(0,0) = CV_VAR_NUMERICAL;
    var_type.at<unsigned int>(0,1) = CV_VAR_NUMERICAL;
    // define output node as numerical
    var_type.at<unsigned int>(0,2) = CV_VAR_NUMERICAL;

    dtree.train(trainingData, CV_ROW_SAMPLE, trainingClasses, cv::Mat(), cv::Mat(),
        var_type, cv::Mat(), CvDTreeParams());
    cv::Mat predicted(testClasses.rows, 1, CV_32F);
    for (int i = 0; i < testData.rows; i++) {
        const cv::Mat sample = testData.row(i);
        CvDTreeNode* prediction = dtree.predict(sample);
        predicted.at<float>(i, 0) = prediction->value;
    }

    cout << "Accuracy_{TREE} = " << evaluate(predicted, testClasses) << endl;
    plot_binary(testData, predicted, "Predictions tree");
}

int main() {

    cv::Mat trainingData(numTrainingPoints, 2, CV_32FC1);
    cv::Mat testData(numTestPoints, 2, CV_32FC1);

    cv::randu(trainingData, 0, 1);
    cv::randu(testData, 0, 1);

    cv::Mat trainingClasses = labelData(trainingData, eq);
    cv::Mat testClasses = labelData(testData, eq);

    plot_binary(trainingData, trainingClasses, "Training Data");
    plot_binary(testData, testClasses, "Test Data");

    svm(trainingData, trainingClasses, testData, testClasses);
    mlp(trainingData, trainingClasses, testData, testClasses);
    knn(trainingData, trainingClasses, testData, testClasses, 3);
    bayes(trainingData, trainingClasses, testData, testClasses);
    decisiontree(trainingData, trainingClasses, testData, testClasses);

    cv::waitKey();

    return 0;
}

```


References

- [Bis95] BISHOP, C. M.: *Neural Networks for Pattern Recognition*. Oxford : Oxford University Press, 1995
- [CV95] CORTES, Corinna ; VAPNIK, Vladimir: Support-Vector Networks. In: *Machine Learning* Bd. 20, 1995, S. 273–297
- [HSW92] HORNIK, K. ; STINCHCOMBE, M. ; WHITE, H.: Multilayer Feedforward Networks Are Universal Approximators. In: WHITE, Halber (Hrsg.): *Artificial Neural Networks: Approximation and Learning Theory*. Oxford, UK : Blackwell, 1992, S. 12–28
- [MP43] MCCULLOCH, W. ; PITTS, W.: A logical calculus of the ideas immanent in nervous activity. In: *Bulletin of Mathematical Biophysics* 5 (1943), S. 115–133
- [Sar02] SARLE, Warren S.: *comp.ai.neural-nets FAQ, Part 3 of 7: Generalization*. <http://www.faqs.org/faqs/ai-faq/neural-nets/part3/>, 2002
- [VC74] VAPNIK, V. N. ; CHERVONENKIS, A. Y.: *Theory of Pattern Recognition [in Russian]*. USSR : Nauka, 1974
- [Wer94] WERBOS, P.J.: *The Roots of Backpropagation: From ordered derivatives to Neural Networks and Political Forecasting*. New York : John Wiley and Sons, 1994