

Handwritten digits recognition in unstructured scenes

Goals

The goal of the project is to create a solution for handwritten digits recognition in unstructured scenes. This task is commonly performed on structured images in which the position of the text is known beforehand. There are also several possible solutions that can be used for handwritten digits recognition: K-nearest neighbors, SVM, logistic regression, CNN, etc. Due to the limited time available for this project I focused on just one machine learning method – KNN. I also decided to start small and to incrementally add more functionality to the project, from performing handwritten digits recognition in ideal circumstances to increasingly more complicated conditions.

Project command line arguments

```
-h      ... display this help.
-d      ... display MNIST digits.                      default: false
-sd     ... perform the single digit detection test on all samples.
-r      N ... constrain the single digit detection test to N randomly selected test samples.
-st     N ... perform the detection over a synthetic image with N digits.
-k      N ... set the number of K nearest neighbors.    default: 1
-cols  C ... set the width (nr. columns) of the synthetic test image. default: 320
-rows  R ... set the height (nr. rows) of the synthetic test image.  default: 240
-p      ... display predicted values.                  default: false
-b      N ... batch mode execution on synthetic test images.
-s      ... suppress user interaction.
```

Figure 1. Command line arguments.

Inspect the MNIST dataset

Use the '-d' command line argument to display the MNIST test set digits one by one. Press 'Esc' to terminate.

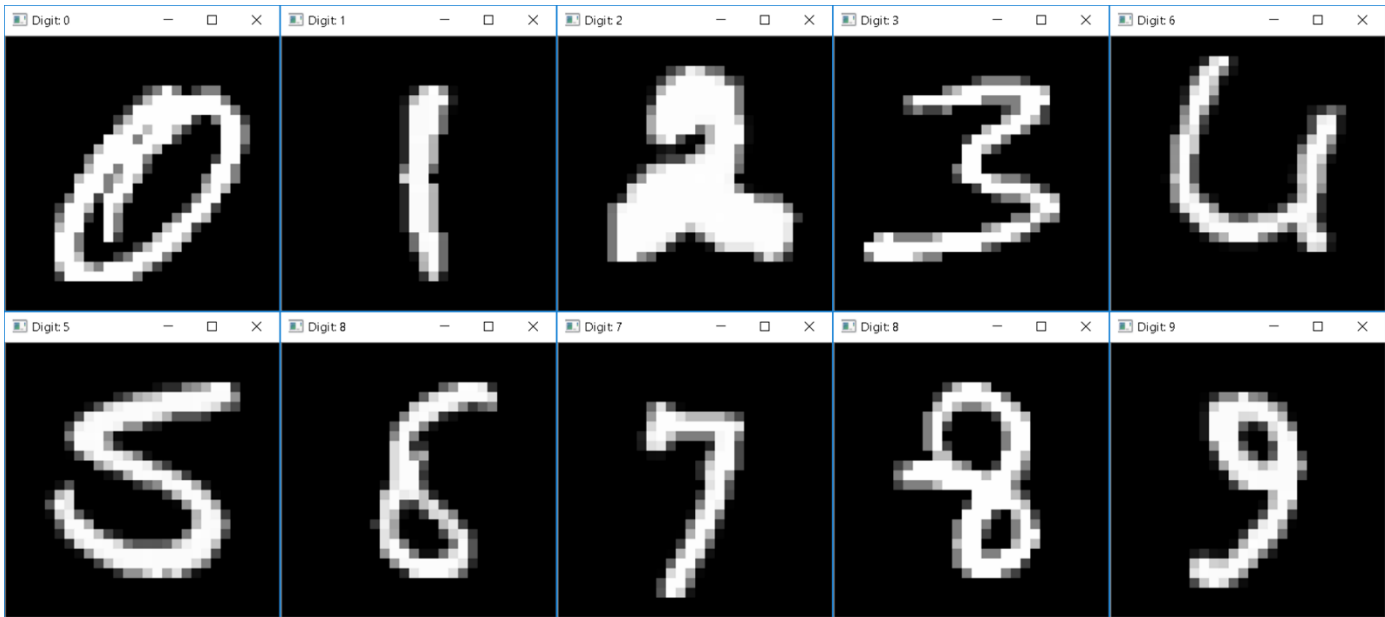


Figure 2. MNIST handwritten digits example.

Single digit detection test

Use the '-sd' command line argument to perform the detection on the entire MNIST test set. Simultaneously use the '-r N' to reduce the test set size to just N randomly picked handwritten digits. At the completion of the test the application will display a summary of the results. For example, in order to run the single digit detection test on 1000 digits use:

```
final.exe -sd -r 1000
```

The output will be something like this:

Accuracy (%):	Test Size:	Value of K:	Duration (ms):	FPS:
96.9	1000	1	9182	109

KNN achieved an accuracy in excess of 95% for all observed cases. In addition the application reports the total execution time of the recognition part of the test and the detection rate (the number of digits per second – FPS).

Use '-k K' to specify the number of neighbors used in the test. From observation there was no real benefit from using K values above 1.

Synthetic test

The next step up in complexity was to perform the detection on a custom test image. The algorithm I used is as follows:

1. Generate a synthetic.
2. Get bounding boxes around potential digits.
3. Filter the bounding boxes by size. Reject small bounding boxes.
4. Fuse any overlapping bounding boxes.
5. Resize & center bounding boxes.
6. Perform the KNN detection test.
7. Display test statistics.

In order to run the test use the following command line arguments:

- '-st N'
perform the detection over a synthetic image with N digits.
- '-k K'
set the number of K nearest neighbors.
- '-b N'
batch mode execution on synthetic test images.

Generate a synthetic image

The custom image was generated by placing randomly selected MNIST digits at random locations on a blank image. The image generation task is handled by the *GetSyntheticImage* method. This method takes as input a test size, the MNIST image and label test sets and returns the synthetic test image (*cv::Mat*) and the labels and locations of the randomly picked (and placed) digits. It was necessary to keep track of the labels and the placement location of the digits in order to be able to automate the evaluation of the detection algorithm.

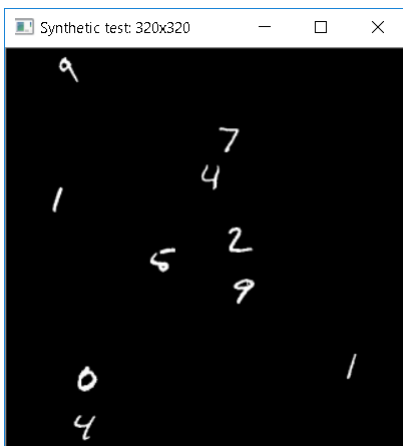


Figure 3. Synthetic image created by randomly placing MNIST digits on a blank image.

Get bounding boxes around potential digits

The goal of this task is to identify the regions in the test image that might contain handwritten digits. The basic steps I used for this purpose are:

1. Perform edge detection
2. Identify contours using *cv::findContours*.
3. Get the bounding boxes around the found contours using *cv::boundingRect*.

Figure 4 shows an example using the Canny edge detector. It turns out that in this case Canny complicated matters and return a much larger number of bounding boxes than actual digits.

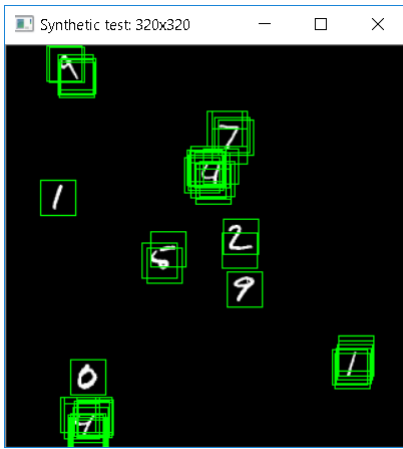


Figure 4. Canny edge detection returns too many bounding boxes.

One potential solution was to perform a smoothing step on the original image before edge detect. In Figure 5 I show the results of applying median blur with four sizes before performing Canny edge detect. The smoothing step did indeed help reduce the number of false positives but also lead to more false negatives.



Figure 5. Median blur 3/5/7/9 before Canny.

`cv::findContours` applied directly on the original image does a great job at capturing all the digits while producing only a small number of false positives.

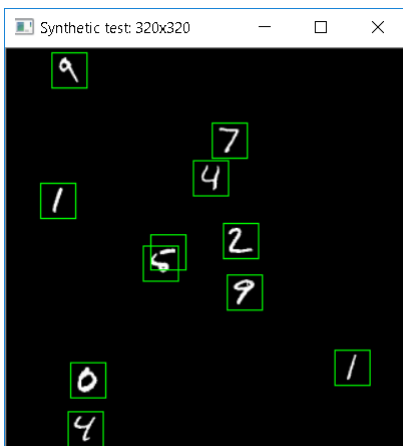


Figure 6. `cv::findContours` directly on the original image.

Filter & Fuse the bounding boxes

Figure 7 shows a few examples of noisy digits that lead to false positives. In order to address this I applied thresholding based on the bounding box size and implemented bounding box fusing.



Figure 7. MNIST data noise produces additional bounding boxes.

The absence of the box filtering and fusing steps (Figure 8) can lead to a drop in accuracy.



Figure 8. False positives in the absence of bounding box filtering and fusing.

Resize & center bounding boxes

One last step was to resize and center the bounding boxes on the found contours in preparation for the detection step.

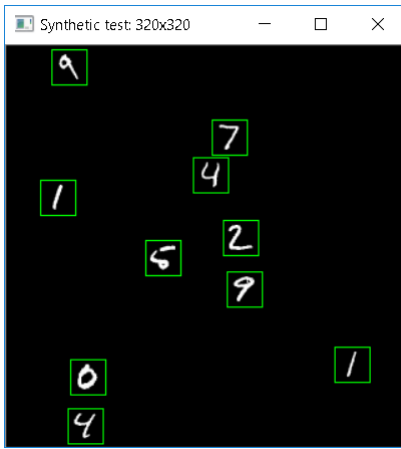


Figure 9. Resize and center.

Perform the KNN detection test

The KNN detection is performed using the OpenCV KNN functionality. My initial implementation was based on OpenCV 2.4 prebuilt binaries. These binaries lacked support for SIMD and multicore processing on the CPU as such the performance (digits per seconds processed) was very low at only **~12 digits per second** in release mode.

Previous attempts at building OpenCV 2.4 with TBB support exposed incompatibilities between the OpenCV 2.4 source and the latest MKL/IPP/TBB libraries from Intel. For this reason I switched to OpenCV 3.1 and was able to build it with support for SSE/AVX, MKL/IPP/TBB.

Using the OpenCV 3.1 binaries the single threaded detection rate increased to **~33 digits per second** in release mode. My multicore based KNN implementation achieved a detection rate of **~120 digits per second** in release mode which represents a **10x** increase relative to the initial single threaded implementation based on OpenCV 2.4.

Display test statistics

On the completion of the test the application reports the test accuracy and duration. The batch option '-b N' allows running multiple tests in sequence and the output statistics look like this:

Accuracy (%)	Test Size	Value of K	Duration (ms)	FPS
78	100	1	894	112
75	100	1	898	111
75	100	1	892	112
72	100	1	913	110
83	100	1	902	111
76	100	1	878	114
76	100	1	900	111
74	100	1	892	112
74.2574	101	1	906	112
79	100	1	900	111

Figure 10. Synthetic image test. Batch mode statistics.

Each test in a batch mode will generate a unique test image.

When selecting a batch size greater than 100 the application will report the detection accuracy mean and standard deviation:

```
Accuracy summary
mean: 77.63%
sigma: 4.016%
```

Figure 11. Detection accuracy mean and standard deviation.

KNearest code – OpenCV 2.4 vs. OpenCV 3.1

Table 1 highlights in **yellow** the mandatory code changes required by the move from OpenCV 2.4 to OpenCV 3.1.

Table 1. KNN code changes when transitioning from OpenCV 2.4 to OpenCV 3.1.

OpenCV 2.4 KNearest example	OpenCV 3.1 KNearest example without multithreading
// nr. of closest neighbors to check out	// nr. of closest neighbors to check out
int K = 1;	int K = 1;
KNearest knn(trnImages, trnLabels, Mat(), false, K);	Ptr<TrainData> trainingData;
	Ptr<KNearest> knn = KNearest::create();
	trainingData = TrainData::create(trnImages, SampleTypes::ROW_SAMPLE, trnLabels);
	knn->setIsClassifier(true);
	knn->setAlgorithmType(KNearest::Types::BRUTE_FORCE);
	knn->setDefaultK(K);
	knn->train(trainingData);
// total number of random test samples used in experiments	// total number of random test samples used in experiments
int testSize = 100;	int testSize = 100;
int truePositives = 0;	int truePositives = 0;
int trueNegatives = 0;	int trueNegatives = 0;
for (int i = 0; i < testSize; i++)	for (int i = 0; i < testSize; i++)
{	{
// pick a random test sample	// pick a random test sample
int index = rand(tstImages.rows);	int index = rand(tstImages.rows);
	Mat results;
int trueValue = (int)tstLabels.at<float>(index, 0);	int trueValue = (int)tstLabels.at<float>(index, 0);
int predicted = (int)knn.find_nearest(tstImages.row(index), K);	int predicted = (int)knn->findNearest(tstImages.row(index), K, results);
if (predicted == trueValue)	if (predicted == trueValue)
truePositives++;	truePositives++;
else	else
trueNegatives++;	trueNegatives++;
}	}

Table 2. The multithreaded code is hidden behind the KNN class.

OpenCV 3.1 KNearest example with multithreading
// number of closest neighbors to check out
int K = 1;
KNN knn(trnImages, trnLabels, K);
knn.Predict(tstImages, tstLabels);

TBB multithreading

I used TBB in order to implement multithreading. The way to do this is to use the `cv::ParallelLoopBody`–`cv::parallel_for_` pattern:

1. Implement a class that inherits the `cv::ParallelLoopBody` class.
In my case this class is called `KNN_Task` and is responsible for single threaded detection of digits.
2. Use the `cv::parallel_for_` construct to automatically partition the detection problem into multiple problems which are executed in parallel.
In my case this was handled inside the `KNN` class (Table 2). Although the multithreading code changes where extensive in the end they resulted in a much cleaner and more readable implementation.

```
// KNN single thread
class KNN_Task : public cv::ParallelLoopBody
{
public:
    KNN_Task
    (
        const KNN *const knn,
        const cv::Mat& tstImages,
        int32_t *const predicted
    ) :
        knn(knn),
        tstImages(tstImages),
        predicted(predicted) {}

private:
    const KNN *const knn;
    cv::Mat tstImages;
    int32_t *const predicted;

public:
    // Single thread body.
    virtual void operator()(const cv::Range& range) const;
};
```

Figure 12. `KNN_Task` performs single threaded KNN detection.
Multiple `KNN_Task` instances execute in parallel in order to fully utilize the CPU.

Algorithm evaluation

In the case of the single digit detection test the algorithm evaluation is straight forward. We know what the input digit is, we compare with the predicted value and based on the result we increment either the true positives or true negatives counters.

The evaluation of the synthetic image test is a little more involved because the digits used in the image composition are randomly selected and placed in random locations.

1. Log input digits labels and their locations
2. Compare the bounding boxes with the known locations. To each match we assign a label. Unrecognized bounding boxes are assigned label '-1'.
3. Perform the detection and compare the predicted digits with the expected values. Increment the true positives and true negatives counters accordingly.

As I mentioned previously the detection rate on the single digit test is above 95% which is very high. However the detection rates on the synthetic test images is much lower (Figure 10) with observed accuracy values as low as 59% and as high as 83%. Using the batch mode I was able to collect statistics for a number of cases. Table 3 shows the observed detection rates for a test size of 100 randomly placed digits and K values between 1 and 10.

Table 3. Synthetic image test statistics. Test size is 100. The value of K is varied between 1 and 10.

		Number of K nearest neighbors									
		1	2	3	4	5	6	7	8	9	10
Accuracy	mean	77.46%	73.28%	76.91%	75.43%	76.99%	75.17%	76.76%	75.66%	76.13%	75.60%
	sigma	4.04%	3.75%	4.88%	4.79%	3.76%	4.72%	4.15%	4.43%	3.71%	4.51%

Detection rates are pretty good but lower than observed for the single digit detection test. The result is surprising because the exact same test digits are used in both the single digit test and the synthetic image test. One hypothesis is that this is caused by the fact that in my KNN implementation I am using the actual pixels as input for the training and prediction. It is conceivable that small shifts in the positions of the digits inside the 28x28 bounding boxes might lead to this accuracy drop.

Conclusion and future work

OpenCV provides a reach set of tools for the rapid prototyping of computer vision and machine learning application. In this project I was able to use these tools to implement a first solution for handwritten digits recognition in unstructured scenes. While the single digit detection rates where very high (>95%) the additional complexity of an unstructured scene lead to substantially lower detection rates (Table 3). This suggests the need to investigate other machine learning algorithms which use features invariant to translation an affine transformations instead of the raw pixel information.

Appendix - OpenCV 3.1 build from source guide

Motivation

The motivation behind using OpenCV 3.1 is basically speed. I have implemented a small sample for handwritten digits recognition (MNIST) using the prebuilt OpenCV 2.4 libraries and I achieved **3FPS** – that is 3 digits per second – in *Debug* mode. Certain conflicts prevent using OpenCV 2.4 source code in conjunction with the latest Intel IPP and MKL libraries however OpenCV 3.1 can use both just fine. In addition OpenCV 3.1 makes the use of OpenCL more transparent than OpenCV 2.4. While Intel IPP and MKL provide primitives optimized for the Intel CPU, OpenCL allows accessing the Intel GPU which on my laptop is about 4x more powerful (in terms of raw GFLOPs) than the CPU.

OpenCV 3.1 build from source guide

Download the source code from here:

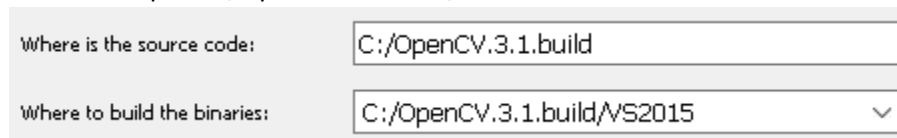
1. OpenCV source code
<https://github.com/opencv/opencv>
2. OpenCV non-free (currently called contrib)
https://github.com/opencv/opencv_contrib

Unzip OpenCV source code in: *C:/OpenCV.3.1*.

Unzip OpenCV contrib and copy folders *doc*, *modules* and *samples* to: *C:/OpenCV.3.1*.

Start CMake and set the input (source code) and output (Visual Studio solution files and output binaries) folders:

- Input: *C:/OpenCV.3.1.build*
- Output: *C:/OpenCV.3.1.build/VS2015*



The image shows a screenshot of the CMake GUI. It has two input fields. The first field is labeled 'Where is the source code:' and contains the text 'C:/OpenCV.3.1.build'. The second field is labeled 'Where to build the binaries:' and contains the text 'C:/OpenCV.3.1.build/VS2015' with a dropdown arrow on the right.

Hit *Configure* for the first time. This will list all the available options.

- > Ungrouped Entries
- > BLAS
- > BUILD
- > CLAMDBLAS
- > CLAMDFFT
- > CMAKE
- > CPACK
- > CUDA
- > Caffe
- > DOXYGEN
- > ENABLE
- > GLIB
- > GSTREAMER
- > Glog
- > HDF5
- > INSTALL
- > JAVA
- > MATLAB
- > OPENCV
- > PYTHON2
- > PYTHON3
- > Tesseract
- > WITH
- > opencv

Disable *WITH_CUDA* and *WITH_CUFFT* (unless you have an NVIDIA GPU installed).

Disable *WITH_OPENCVCLAMDBLAS* and *WITH_OPENCVCLAMDFFT* (unless you have an AMD GPU installed).

Enable *WITH_OPENCV*. OpenCV 3.1 will automatically call OpenCL functions when possible and makes sense from the performance point of view. Additional information at: <http://opencv.org/platforms/opencv.html>.

By default CMake will use the OpenCL 1.2 header files. If you have the Intel OpenCL SDK you can tell CMake where to find it:

- *OPENCV_INCLUDE_DIR*: <path to Intel OpenCL headers>
C:/Program Files (x86)/Intel/OpenCL SDK/6.3/include
- *OPENCV_LIBRARY*: <path to Intel OpenCL.lib>

I had to disable *WITH_MATLAB*. I am configuring OpenCV for x86 and my Matlab installation is x64.

OpenCV 3.1 requires Python 3. CMake might not find your Python installation.

In my case I had to tell CMake where to find Python 3.5.2.

PYTHON3	
PYTHON3_EXECUTABLE	C:/Python/3.5.2/python.exe
PYTHON3_INCLUDE_DIR	C:/Python/3.5.2/include
PYTHON3_INCLUDE_DIR2	
PYTHON3_LIBRARY	C:/Python/3.5.2/libs/python35.lib
PYTHON3_LIBRARY_DEBUG	
PYTHON3_NUMPY_INCLUDE_DIRS	
PYTHON3_PACKAGES_PATH	C:/Python/3.5.2/Lib/site-packages

Additional settings might be required to full enable Python.

Enable *OPENCV_ENABLE_NONFREE*.

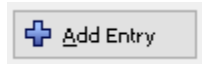
▼ OPENCV	
OPENCV_CONFIG_FILE_INCLUDE_DIR	C:/OpenCV.3.1.build/VS2015
OPENCV_ENABLE_NONFREE	<input checked="" type="checkbox"/>
OPENCV_EXTRA_MODULES_PATH	
OPENCV_WARNINGS_ARE_ERRORS	<input type="checkbox"/>

I have Intel MKL, IPP and TBB already installed (free Community License).

Disable *WITH_TBB* for now (or skip to the end when I enable it).

Disable *WITH_IPP_A*. Apparently this is a preview library and not part of the freely available IPP package.

The option *WITH_IPP* will cause CMake to download IPP ICV version. In order to use the installed IPP we need to tell CMake where to find it.



Click *Add Entry* and specify:

- Name: *IPPROOT*
- Type: *PATH*
- Value: <path to IPP>
C:/Program Files (x86)/IntelSWTools/compilers_and_libraries/windows/ipp

The Intel MKL settings are found under the *BLAS*, *LAPACK* and *Ungrouped Entries* sections. Not all of the following settings are available at first. Clicking *Configure* after adding the available settings will cause CMake to reveal new settings.

- *BLAS_mkl_intel_c_LIBRARY*
C:/Program Files (x86)/IntelSWTools/compilers_and_libraries/windows/mkl/lib/ia32/mkl_intel_c.lib
- *BLAS_mkl_intel_ip64_LIBRARY*
C:/Program Files (x86)/IntelSWTools/compilers_and_libraries/windows/mkl/lib/intel64/mkl_intel_ip64.lib
- *BLAS_blas_LIBRARY*
C:/Program Files (x86)/IntelSWTools/compilers_and_libraries/windows/mkl/lib/ia32/mkl_blas95.lib
- *BLAS_mkl_sequential_LIBRARY*
C:/Program Files (x86)/IntelSWTools/compilers_and_libraries/windows/mkl/lib/ia32/mkl_sequential.lib
- *BLAS_libiomp5md_LIBRARY*
C:/Program Files (x86)/IntelSWTools/compilers_and_libraries/windows/compiler/lib/ia32/libiomp5md.lib
- *BLAS_mkl_core_LIBRARY*
C:/Program Files (x86)/IntelSWTools/compilers_and_libraries/windows/mkl/lib/ia32/mkl_core.lib
- *BLAS_mkl_intel_thread_LIBRARY*
C:/Program Files (x86)/IntelSWTools/compilers_and_libraries/windows/mkl/lib/ia32/mkl_intel_thread.lib
- *LAPACK_lapack_LIBRARY*
C:/Program Files (x86)/IntelSWTools/compilers_and_libraries/windows/mkl/lib/ia32/mkl_lapack95.lib
- *MKL_LAPACK_INCLUDE_DIR*
C:/Program Files (x86)/IntelSWTools/compilers_and_libraries/windows/mkl/include
- *MKL_CBLAS_INCLUDE_DIR*
C:/Program Files (x86)/IntelSWTools/compilers_and_libraries/windows/mkl/include

Enable:

- AVX
- AVX2
- SSE41
- SSE42
- SSSE3

▼ ENABLE

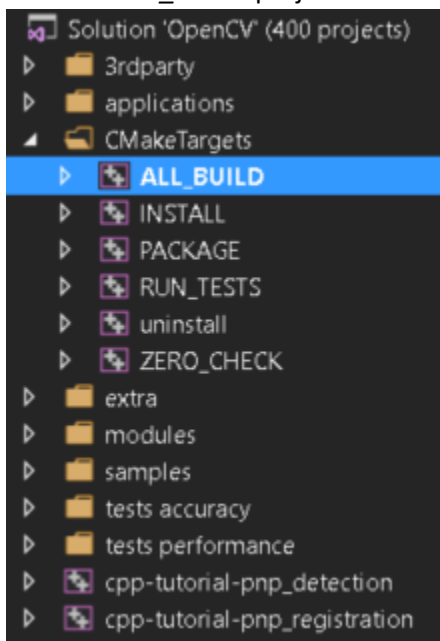
ENABLE_AVX	<input checked="" type="checkbox"/>
ENABLE_AVX2	<input checked="" type="checkbox"/>
ENABLE_CCACHE	<input type="checkbox"/>
ENABLE_FMA3	<input type="checkbox"/>
ENABLE_IMPL_COLLECTION	<input type="checkbox"/>
ENABLE_INSTRUMENTATION	<input type="checkbox"/>
ENABLE_NOISY_WARNINGS	<input type="checkbox"/>
ENABLE_POPCNT	<input type="checkbox"/>
ENABLE_PRECOMPILED_HEADERS	<input checked="" type="checkbox"/>
ENABLE_SOLUTION_FOLDERS	<input checked="" type="checkbox"/>
ENABLE_SSE	<input checked="" type="checkbox"/>
ENABLE_SSE2	<input checked="" type="checkbox"/>
ENABLE_SSE3	<input checked="" type="checkbox"/>
ENABLE_SSE41	<input checked="" type="checkbox"/>
ENABLE_SSE42	<input checked="" type="checkbox"/>
ENABLE_SSSE3	<input checked="" type="checkbox"/>

Enable `BUILD_EXAMPLES`. This will generate the OpenCV example projects which are useful when trying to understand how a certain functionality works.

Click *Generate*.

Open the Visual Studio solution.

Build the `ALL_BUILD` project.

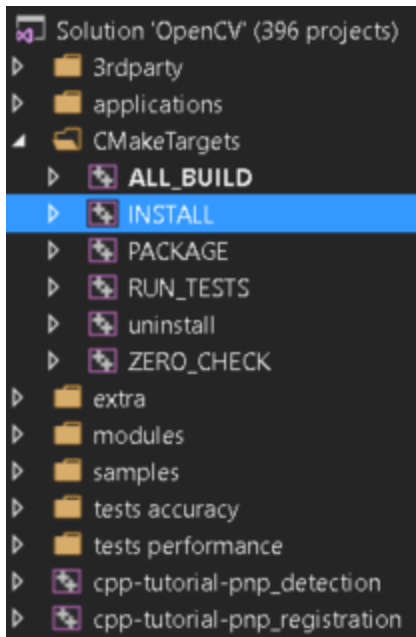


Right away the `opencv_core` project fails to build due to a DirectX conflict. You can stop the build.

In CMake disable `WITH_DIRECTX`. *Configure. Generate*.

The `ALL_BUILD` project builds just fine now.

In order collect the OpenCV built files in one place build the `INSTALL` project.



This will copy the OpenCV built files to the folder *C:/OpenCV.3.1.build/VS2015/install*.

Build the *ALL_BUILD* and *INSTALL* projects for both *Debug* and *Release* configurations.

Copy *etc*, *include* and *x86* from *C:/OpenCV.3.1.build/VS2015/install/* to *C:/OpenCV.3.1/build*.

The *opencv_nonfree310.lib* is replaced with *opencv_xfeatures2d310.lib* in OpenCV 3.1. Update this in Project Properties -> Linker -> Input -> Additional Dependencies.

Initial results

Using the newly built OpenCV 3.1 binaries my KNN sample (which needed updating to the new API) ran at about **10FPS** or **3x** faster than with OpenCV 2.4 (this was measured with the *Debug* configuration). Switching to the *Release* configuration my sample achieved **~33FPS** with the OpenCV 3.1 binaries. This is due to the use of IPP, MKL and SSE/AVX extensions for the CPU. However there's still room for improvement – these experiments were performed on a single CPU core.

Multithreading

IPP

By default the IPP installer does not install the multithreaded libraries. I modified the existing installation and added all the available options. Even so CMake still complains about missing *ippmmt.lib*.

TBB

Set the following CMake variables:

- **TBB_INCLUDE_DIRS**
C:/Program Files (x86)/IntelSWTools/compilers_and_libraries/windows/tbb/include
- **TBB_LIB_DIR**
C:/Program Files (x86)/IntelSWTools/compilers_and_libraries/windows/tbb/lib/ia32/vc14
- **TBB_STDDEF_PATH**
C:/Program Files (x86)/IntelSWTools/compilers_and_libraries/windows/tbb/include/tbb/tbb_stddef.h

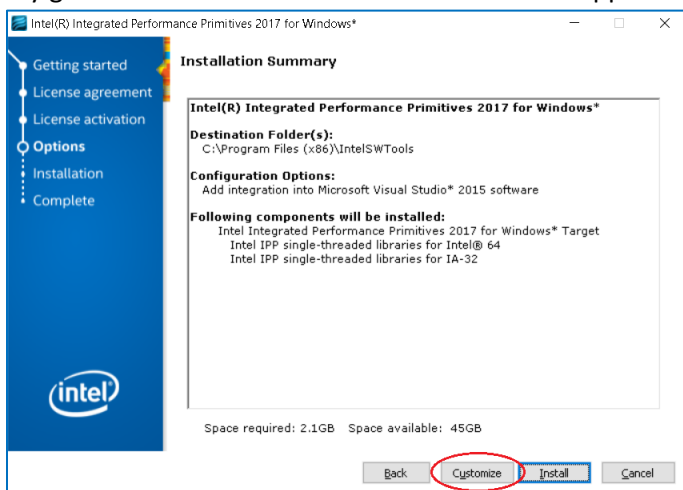
Disable **WITH_OPENMP**. Apparently OpenMP and TBB are mutually exclusive.

CMake reports: *Parallel framework: TBB (ver 2017.0 interface 9102)*.

My measurements show no difference with or without TBB. Additional code is required to parallelize my sample and fully utilize the CPU cores. Potential solutions:

- [Multithreading in OpenCV using TBB](#)
- [OpenCV and TBB in image processing](#)

Regarding the missing *ippmmt.lib* library, one [link](#) suggests reinstalling IPP. Having done so CMake is still complaining. My guess is that the installer is broken and the supposed “hidden” options are missing.



OpenCL

In order to test the OpenCL backend we need to make a few relatively small changes in the code. The key is to use the UMat structure instead of Mat ([OpenCV T-API](#)). At the moment however I didn't notice a change in performance. This could mean one of several things. For example:

- my code changes fail to invoke OpenCL
- OpenCV 3.1 lacks the OpenCL primitives required by my sample and falls back to the CPU

- OpenCV 3.1 OpenCL implementation is suboptimal

OpenCV multithreading

OpenCV offers support for parallelizing code across multiple cores.

The user is required to follow this guideline:

- Create a class wrapper for the code that needs parallelizing.
- Inherit this class from `cv::ParallelLoopBody`.
- Implement the `virtual void operator()(const cv::Range& range) const` method.
This method is the thread body and performs useful work on a subset of the whole problem.
- Use the `cv::parallel_for_` construct to launch multiple threads in parallel.

I created a class called *KNN* to represent the total amount of work to be performed and a class called *KNN_Task* which represents the worker thread. *KNN* calls `cv::parallel_for_` which in turn invokes *KNN_Task* multiple times in parallel. My sample is now about **4x** faster than without multithreading (I am running the tests on a **4-core** Intel CPU).

My original implementation for OpenCV 2.4 was achieving **3FPS** (digits / second) in *Debug* mode. The version which uses OpenCV 3.1 with multithreading is achieving **~33FPS** in *Debug* mode and **~120FPS** in *Release* mode.