eCognition Developer

Tutorial 7 — Convolutional Neural Networks in eCognition

www.trimble.com

Introduction

About this Tutorial

Artificial neural networks have long been popular in machine learning. More recently, they have received renewed interest, since networks with many layers (often referred to as deep networks) have been shown to solve many practical tasks with accuracy levels not yet reached with other machine learning approaches. In image analysis, **convolutional neural networks** have been particularly successful. The term refers to a class of neural networks with a specific network architecture, where each so-called **hidden layer** typically has two distinct stages: the first stage is the result of a **local convolution** of the previous layer (the kernel has trainable weights), the second stage is a **max-pooling** stage, where the number of units is significantly reduced by keeping only the maximum response of several units of the first stage. After several hidden layers, the final layer is typically a fully connected layer. It has a unit for each class that the network predicts, and each of those units receives input from all units of the previous layer.

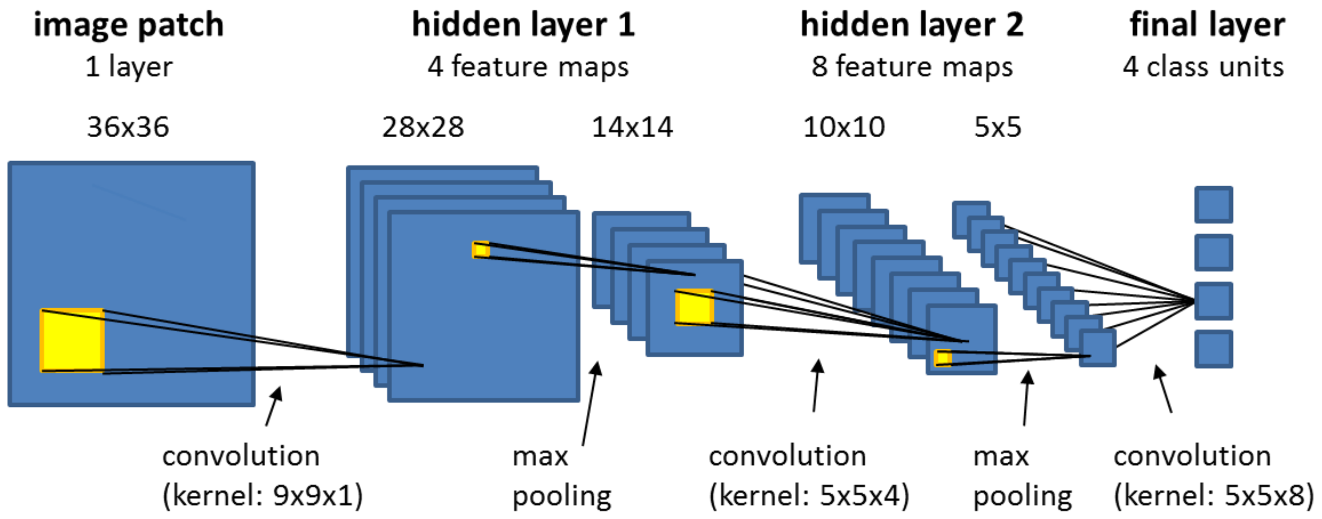


Figure 0.1: *Schematic representation of a convolutional neural network with two hidden layers. Kernel weights are optimized during training with labeled samples, i.e., image patches whose class is known.*

This tutorial gives you an introduction to using convolutional neural networks in eCognition, which is based on the Google TensorFlow™ API. In particular, you will learn how to create, train, and apply convolutional neural networks. This tutorial assumes basic knowledge of the eCognition Developer software.

This module has two lessons:

- Lesson 1 shows you how to use the new algorithms for convolutional neural networks
- Lesson 2 shows you how to use the model output layer to find objects and evaluate the model accuracy

Further information about eCognition products is available on our website:

www.eCognition.com

Requirements

To perform this Tutorial, you will need:

- **eCognition Developer** version 9.3 (or higher) installed on a computer
- A computer **mouse** is highly recommended

All steps of this tutorial can be done using **eCognition Developer.** You can also use the free-trial version, however, in this case, you won't be able to save your network or export the results.

This tutorial is designed for self-study.

Data included with the Tutorial

The tutorial folder contains:

- an image **M-31-01.jpg**, representing a Russian topographic map available for download on the internet
- a **GroundTruth.shp**, which denotes the locations of the symbol for church in the map (manually created)
- a rule set **Tutorial.dcp**

Lesson 1 – First steps with convolutional neural networks

1.0 Lesson Content

This lesson gives you a first taste of the steps involved in using convolutional networks for practical applications. The task we set ourselves is to find cross-like symbols, representing churches, on a Russian topographical map of London, which is available for free on the internet (see Figure 1.1).

To evaluate the neural network approach, we follow common machine-learning practice: we train the model on a training data set (training region), and validate the model on a test data set (test region) that has not been used during training.

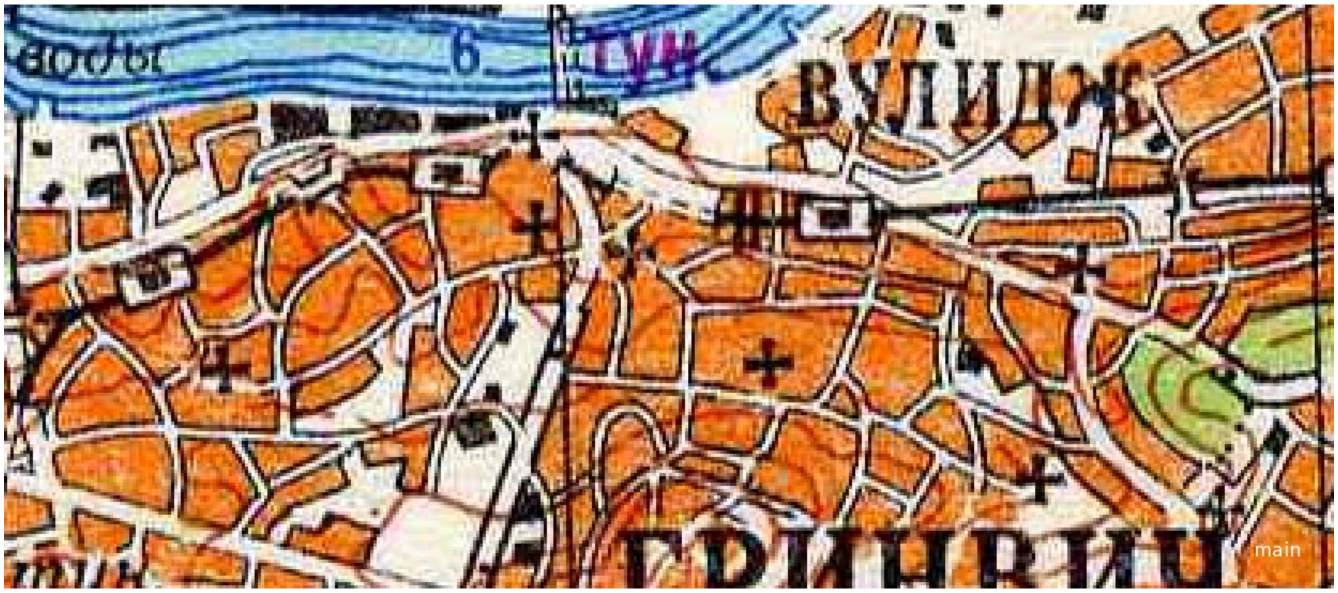


Figure 1.1: *A subset of a topographical map of London. Our goal is to detect the cross symbols.*

In this lesson, you will learn to:

1. Create a project and **classify samples for targets and non-targets** in the training region of the image.
2. Use the algorithm *generate labeled sample patches* to **store sample image patches** on the hard disk for later use in training.
3. **Create a convolutional network** with a single hidden layer using the algorithm *create convolutional neural network*.
4. Use the algorithm *train convolutional neural network* and the collected samples to **train the network**, i.e. adjust its weights to optimize classification accuracy of the samples.
5. **Apply the network** by using the algorithm *apply convolutional neural network* to a **test region** and generate a heatmap layer for your target class. Values close to one reflect high evidence for the class, and values close to zero reflect low evidence.
6. **Save your trained model** using *save convolutional neural network* algorithm.

1.1 Create a project and prepare a classified image object level

Start **eCognition Developer**.

Via the menu File > Load image file: load the image in the tutorial folder.

Via the menu Process > Load rule set: load the ruleset in the tutorial folder.

Execute the following processes step-by-step:

- **load ground truth**: this loads the shapefile from the tutorial folder and shows the ground truth in yellow (see Figure 1.2a).
- **define regions**: defines the map region, from which samples are selected, and a test region. Samples in the test region will be discarded and are not used for training. The test region will later be used for model validation.
- **create class Target**: classifies circular regions (radius 3 pixels) around each target location in the map region as class *Target* (see Figure 1.2b). Making the targets larger than a single pixel has two advantages: there are more samples available to train the network (1 sample for each classified pixel), and our model will learn and create a target heatmap that shows high values in these circular regions around the target center (and not just at a single target pixel), leading to a more robust target detection.

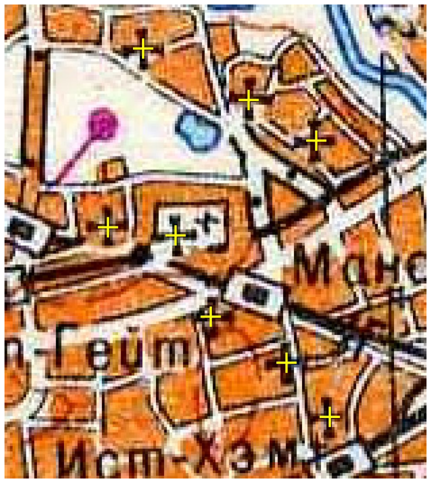


Figure 1.2a: *Yellow crosses denote the ground truth layer.*

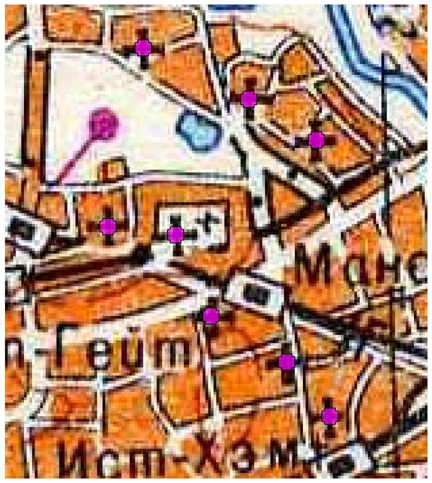


Figure 1.2b: *Targets are classified in pink.*

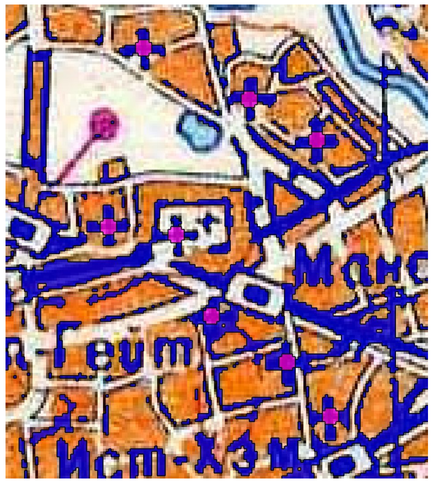


Figure 1.2c: *Non-Targets are classified in blue.*

- **create class Non-Target**: classifies dark areas in the map region as class *Non-Target* (see Figure 1.2c). To train the neural network, we need samples of at least two classes. We could simply classify all unclassified objects as Non-Target, but we classify only dark pixels here, expecting those to be the most likely false positives, as they are similar to the symbol we want the network to learn. To identify dark regions, a gaussian smoothing is performed on a Brightness layer, followed by a threshold segmentation.

- **remove classification from test region**: removes all classified objects from the test region.
- **evenly sample objects**: performs a chessboard segmentation (size 1) on the classified objects. Each of the pixel-sized objects now has an equal chance to be picked up as a sample in the subsequent analysis.

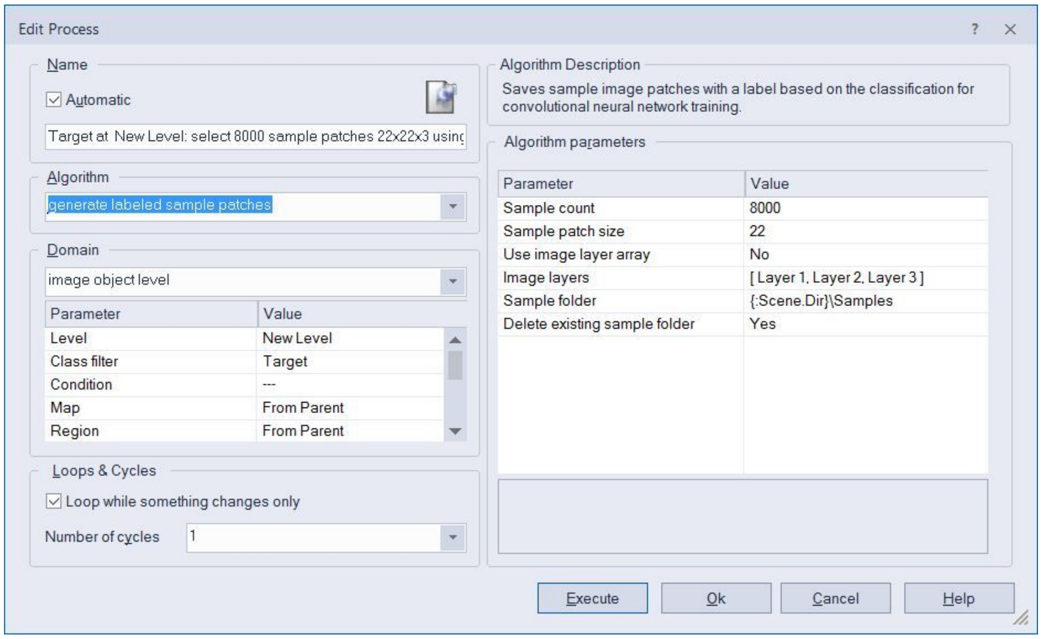
1.2 Create labeled sample patches

Now you are ready to create sample patches that can be used for training a convolutional neural network in eCognition. To do this you need the eCognition algorithm *generate labeled sample patches*,

- Edit the first process in "CREATE SAMPLES" by clicking on it, and selecting edit from the context menu (see Figure 1.3).

This algorithm will create 8000 sample patches, randomly picked from all the pixels of the image object domain (here: all Target objects). For each pixel that gets picked by the algorithm, an image patch of 22x22 pixels around the central pixel is exported to the Sample folder. These samples have three layers (Layer 1, Layer 2, and Layer 3).

The "Delete existing sample folder" option has been set to "yes", meaning that any samples already in the folder are deleted. This is good practice for the first time you add samples to the sample folder, as it will ensure that the rule set behaves the same way every time you execute it, and does not accumulate more and more samples. (Those samples may also be outdated if you were to make changes to the rule set.) You need to be careful though when using this option: make sure your sample folder is specified correctly, to avoid an unpleasant surprise when you find that valuable data was accidentally deleted.



Name
☑ Automatic
Target at New Level: select 8000 sample patches 22x22x3 usinç

Algorithm: generate labeled sample patches

Domain: image object level

Parameter	Value
Level	New Level
Class filter	Target
Condition	---
Map	From Parent
Region	From Parent

Loops & Cycles
☑ Loop while something changes only
Number of cycles: 1

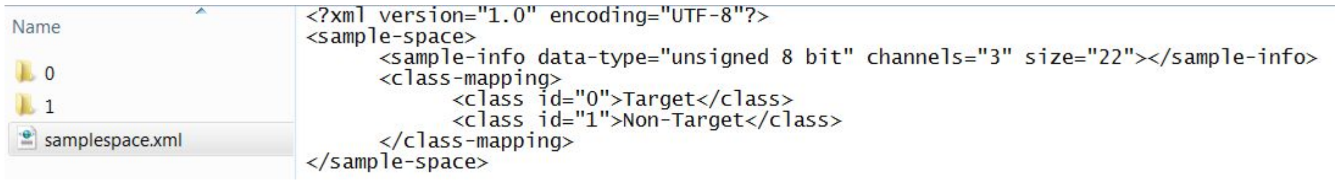
Algorithm Description
Saves sample image patches with a label based on the classification for convolutional neural network training.

Algorithm parameters

Parameter	Value
Sample count	8000
Sample patch size	22
Use image layer array	No
Image layers	[Layer 1, Layer 2, Layer 3]
Sample folder	{:Scene.Dir}\Samples
Delete existing sample folder	Yes

Figure 1.3: *Settings of algorithm: generate labeled sample patches.*

- **execute the CREATE SAMPLES process** with both subprocesses. This will take a few minutes.
- verify that image patches have indeed been exported to the sample folder. Samples of different classes are exported to different folders. The *samplespace.xml* stores additional information needed for eCognition, and should not be manipulated (see Figure 1.4).



```
<?xml version="1.0" encoding="UTF-8"?>
<sample-space>
      <sample-info data-type="unsigned 8 bit" channels="3" size="22"></sample-info>
      <class-mapping>
            <class id="0">Target</class>
            <class id="1">Non-Target</class>
      </class-mapping>
</sample-space>
```

Figure 1.4: *The contents of the sample folder viewed in the explorer. Samples for different class labels are divided into different folders. The samplespace.xml is used by eCognition to store impportant information.*

The larger number of Non-Target samples was chosen here because the image contains much more Non-Target than Target regions, thus misclassifications of Non-Targets are more problematic. The idea is to have this bias reflected during the training, while still providing a good mixture of Targets and Non-Targets.

1.3 Create a convolutional neural network

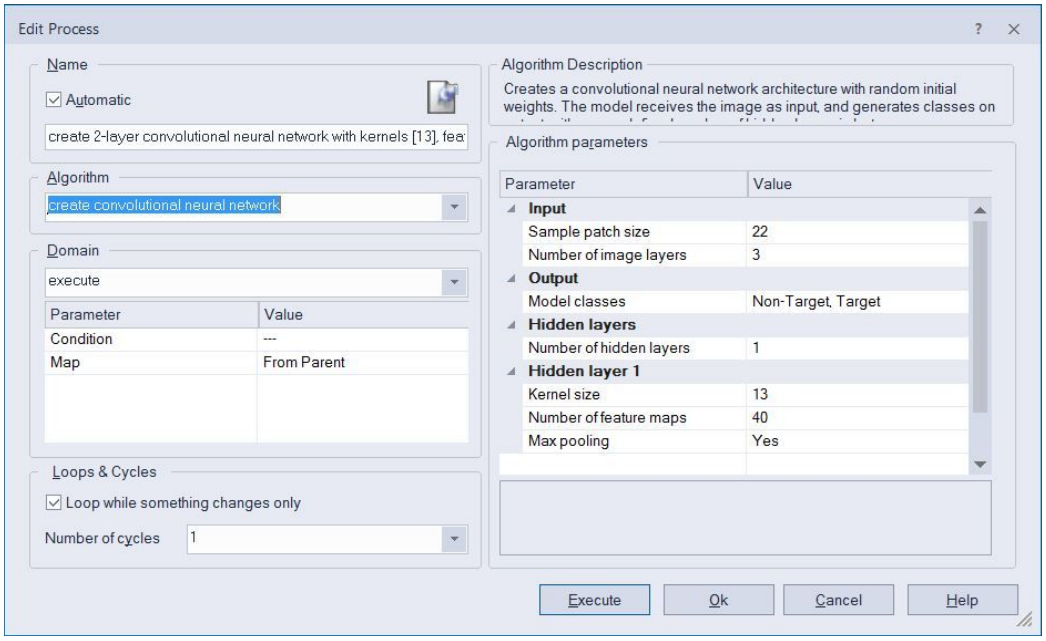
In the next step, we create a convolutional neural network. This requires only one algorithm.

- Edit the process 'Create convolutional neural network' and check the settings (see Figure 1.5).

The algorithm defines the size of the samples (here 22x22 pixels, 3 layers) that will be fed into the model, and the classes generated on output (here: Target and Non-Target).

It also defines the number of hidden network layers (here: 1). For each layer, the size of the convolutional kernel needs to be specified, as well as the number of features to be generated, and whether to use spatial pooling or not. The network defined here uses only one hidden layer:

- The kernel size is 13x13.
- 40 feature maps are generated.
- spatial max pooling is performed



☑ Automatic

create 2-layer convolutional neural network with kernels [13], fea

Algorithm: create convolutional neural network

Domain: execute

Parameter	Value
Condition	---
Map	From Parent

☑ Loop while something changes only

Number of cycles: 1

Creates a convolutional neural network architecture with random initial weights. The model receives the image as input, and generates classes on

Parameter	Value
Input	
Sample patch size	22
Number of image layers	3
Output	
Model classes	Non-Target, Target
Hidden layers	
Number of hidden layers	1
Hidden layer 1	
Kernel size	13
Number of feature maps	40
Max pooling	Yes

Figure 1.5: *Settings of algorithm: create convolutional neural network.*

The **hidden layer kernel thus corresponds to 3x13x13x40 weights**. The first factor corresponds to the number of feature maps in the previous hidden layer (or here, the image layers) and the second and third factors describe the number of units in the local neighborhood, from which connections are formed into the hidden layer. The final factor corresponds to the number of feature maps generated. After all, we do not train only one kernel of size 3x13x13, but 40 different ones. The only hidden layer of this network thus contains 20,280 different weights, which can be trained.

Note that the **spatial extent of the feature maps shrinks** from layer to layer (see Figure 1.6). While the original sample input has 22x22 units, after convolution with a 13x13 kernel only 10x10 valid units remain (units further to the side are receiving non-existing input, and are dropped from the network). After max pooling, the network layer further shrinks by a factor of two in both dimensions, we thus have only 5x5(x40) units left. The final layer in this network has two units left: one for class Target, one for class Non-Target, and both connect to all 1000 units of the previous layer (reflecting another 2000 weights to be trained).

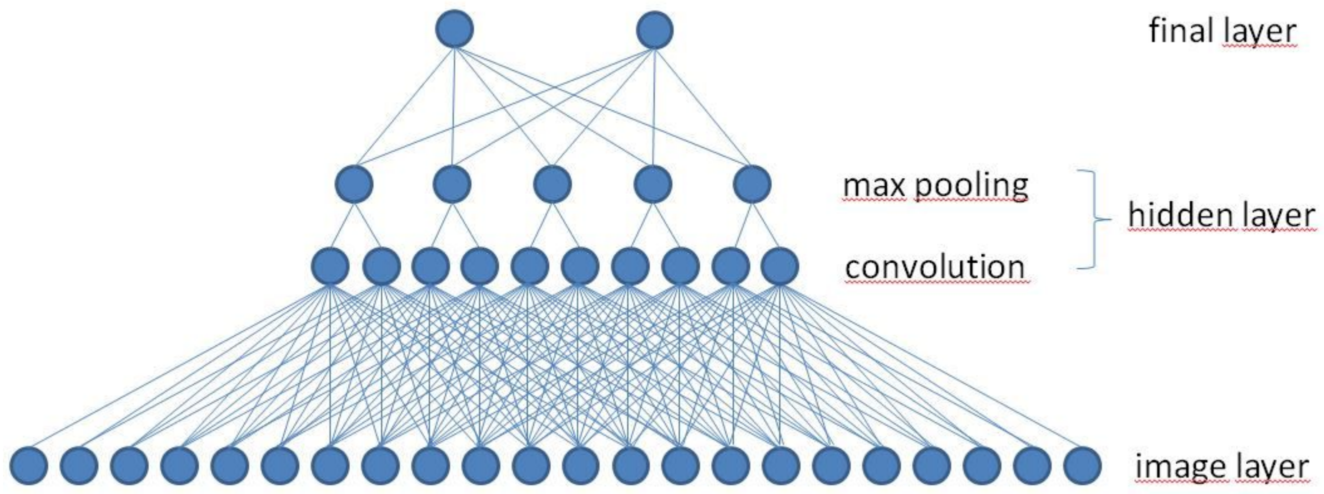


Figure 1.6: *A simplified schematic representation of the model. The image layer of the sample patch has 22 units (in reality, there are 22x22x3 units). After convolution with a kernel of size 13 we obtain a valid feature map with 10 units, which after max pooling, is reduced to 5 units. (In reality, the feature map has 40 distinct layers, not just one, and is, of course, also two dimensional). The final layer has two units, which represent the two trained classes. Each is connected to all units in the max pooling stage of the hidden layer.*

We **strongly advise you to work only with odd sized kernels** (e.g. 13x13, and not 12x12) as even sized kernels will generate hidden layer units located "between pixels", and then are shifted slightly to match pixel borders. In fact, it is a good idea to **track the number of units from layer to layer**, to ensure that max pooling makes sense (you need an even number of units to do max pooling) and to ensure that the fully connected final layer, whose kernel size is automatically adjusted by the software, also has an odd sized number of units.

To give an example: if you had started out here with a sample size of 21x21 and performed a convolution with a 13x13 kernel, you would end up with 9x9 units, and cannot perform a valid max-pooling. If you started out with 20x20 pixels, you end up with 8x8 units and can perform a valid max-pooling, resulting in 4x4 pixels. However, if this is your final hidden layer, your last convolution kernel will be 4x4, which is even sized. You can use such a model, but you need to be aware that when you apply your model, the hot spots in your heatmap layer will likely be somewhat shifted with respect to the target locations.

While so-called **deep learning networks** can comprise 100 and more layers with 1000s of feature maps, **we recommend you start experimenting with just a few layers and feature maps**, as we do here. When you add more layers and features, more weights need to be optimized during training, typically requiring many additional samples and longer training times. Moreover, a more complex model may not converge as easily during training.

So while adding layers and features makes your model more powerful in principle, it won't always improve the performance of your trained model, because optimal weights are harder to find. We hope this tutorial will encourage you to play with different settings, perhaps also on your own data, to gain your own experiences.

- **execute the process CREATE MODEL**

Now the model can in principle already be used, but as its weights are set to random values, it will not be useful in practice before it has been trained.

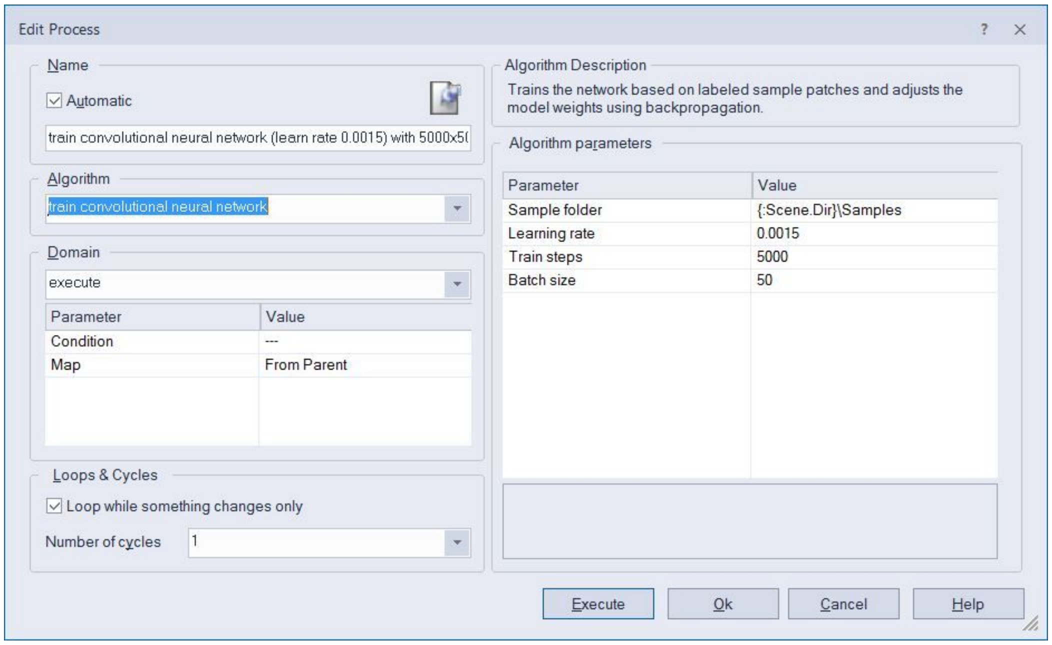
1.4 Train your network

Training consists of many individual training steps. In each step, a randomly selected batch of samples is fed into the model, gradients for each weight are evaluated using backpropagation, and weights are optimized using a statistical gradient descent. Again, a single algorithm takes care of everything.

- edit the process *train convolutional neural network* (see Figure 1.7).

This algorithm requires you to specify:

- the sample folder, which contains the labeled samples for the supervised training
- the learning rate (here 0.0015)
- the number of training steps (here 5000)
- the batch size (number of samples used at each training step, here 50)



Name: train convolutional neural network (learn rate 0.0015) with 5000x5(

Algorithm: train convolutional neural network

Domain: execute

Parameter	Value
Condition	---
Map	From Parent

Loop while something changes only: ✓

Number of cycles: 1

Algorithm Description: Trains the network based on labeled sample patches and adjusts the model weights using backpropagation.

Parameter	Value
Sample folder	{:Scene.Dir}\Samples
Learning rate	0.0015
Train steps	5000
Batch size	50

Figure 1.7: *Settings of the algorithm: train convolutional neural network.*

- **execute the process TRAIN MODEL**. (This will take a few minutes.)

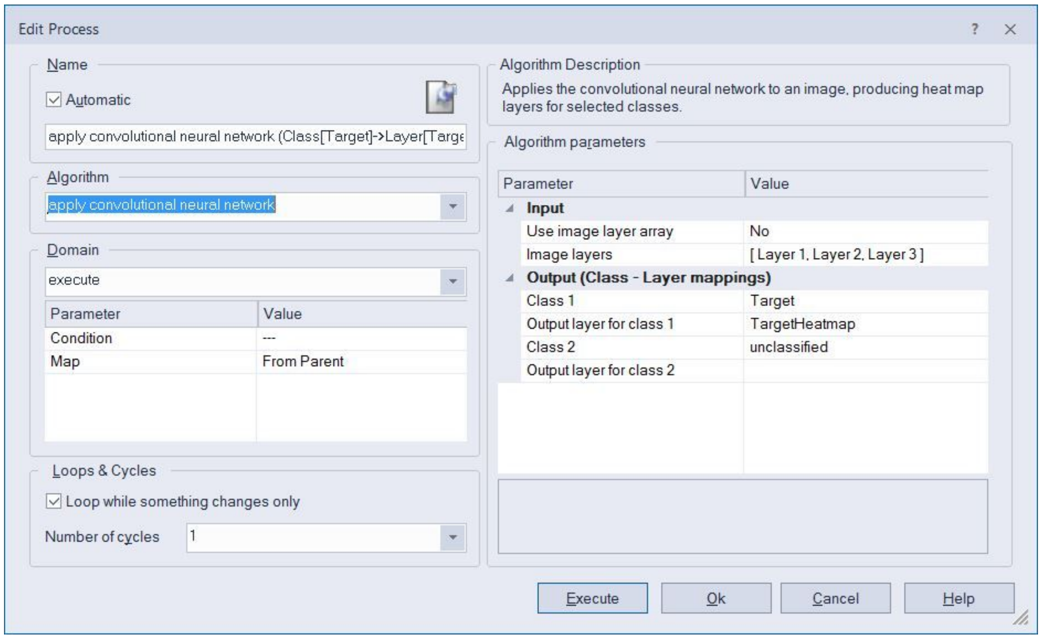
For training, the **learning rate is a particularly important parameter**. It defines the amount by which weights are adjusted at each iteration of the statistical gradient descent optimization used by eCognition. If your learning rate is too small, the learning process is not only slow, but you may get stuck in local minima and end up with weights not even close to the optimal settings. If your learning rate is too large, you may have fast initial improvement of your model, but you may not reach the bottom of the minimum, but rather "jump around it" (because the changes to the weights are too dramatic), or, you may end up with invalid results and your model may produce NaN values.

A gradual decay of the learning rate during learning is common practice (but not implemented in this tutorial).

1.5 Use your network

Finally, we are ready to make use of our network, and discover what it has learned.

- Edit the process *apply convolutional neural network*.



Edit Process

Name

☑ Automatic

apply convolutional neural network (Class[Target]->Layer[Targe

Algorithm: apply convolutional neural network

Domain: execute

Parameter	Value
Condition	---
Map	From Parent

Loops & Cycles

☑ Loop while something changes only

Number of cycles: 1

Algorithm Description

Applies the convolutional neural network to an image, producing heat map layers for selected classes.

Algorithm parameters

Parameter	Value
Input	
Use image layer array	No
Image layers	[Layer 1, Layer 2, Layer 3]
Output (Class - Layer mappings)	
Class 1	Target
Output layer for class 1	TargetHeatmap
Class 2	unclassified
Output layer for class 2	

Figure 1.8: *Settings of the algorithm: apply convolutional neural network.*

This algorithm requires you to specify the following settings:

- image layers used as model input. This corresponds to the layers of the samples, which were used to train the model (here: Layers 1, 2 and 3).
- model class(es) for which to create a heatmap, and a layer to which this heatmap is written. (In this layer, values close to one indicate that the model predicts a high likelihood of target presence, values close to zero indicate a low target likelihood.) Here, we are only interested in the result for the class *Target*, and we want the algorithm to create a raster layer *TargetHeatmap*.
- **execute the process APPLY MODEL** to create and display a map of the TestRegion, and to generate and display the layer TargetHeatmap.

You can see that indeed red dots generally appear in image locations that contain or resemble a target. Although the convolutional neural network contains only one hidden layer, the model has learned successfully after only 5000 training steps.

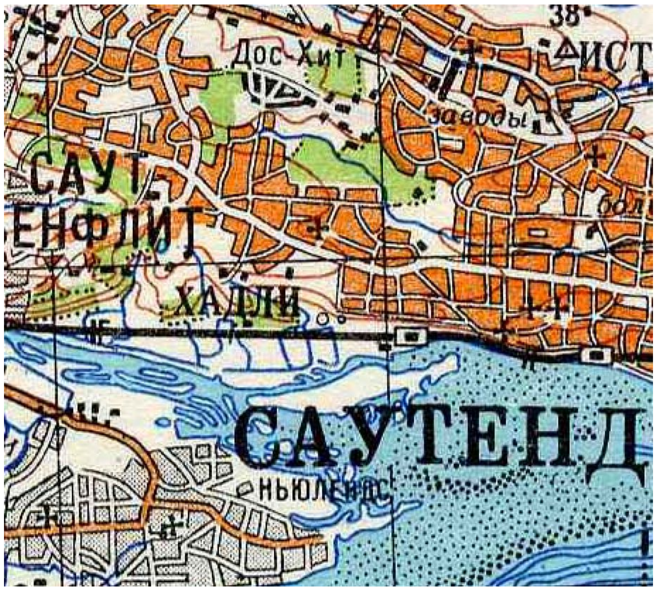


Figure 1.9a: A subset of the test region.

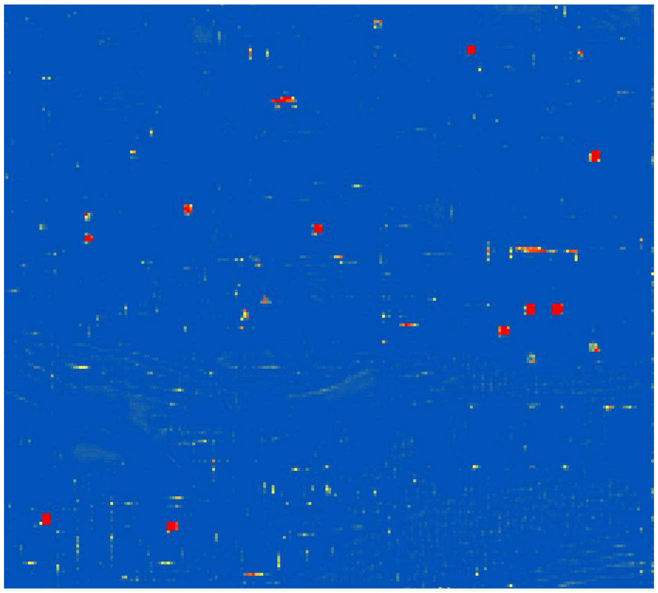
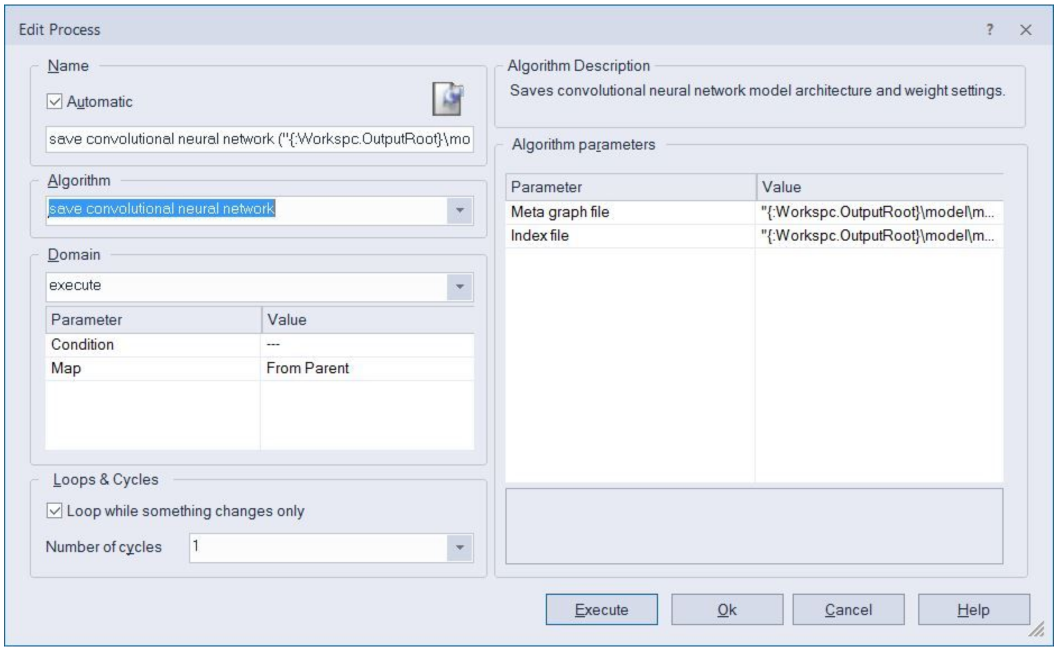


Figure 1.9b: *The generated TargetHeatmap Layer. Red indicates high values close to 1, blue indicates values close to zero.*

1.6 Save your network

As a last step in this first lesson, we will save the network for later use in production mode.

- Edit the algorithm *save convolutional neural network* and verify its settings.



Edit Process

Name

☑ Automatic

save convolutional neural network ("{:Workspc.OutputRoot}\mo

Algorithm

save convolutional neural network

Domain

execute

Parameter	Value
Condition	---
Map	From Parent

Loops & Cycles

☑ Loop while something changes only

Number of cycles: 1

Algorithm Description

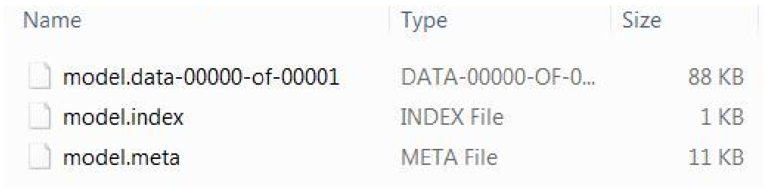
Saves convolutional neural network model architecture and weight settings.

Algorithm parameters

Parameter	Value
Meta graph file	"{:Workspc.OutputRoot}\model\m...
Index file	"{:Workspc.OutputRoot}\model\m...

Figure 1.10: *Settings of the algorithm: save convolutional neural network.*

The algorithm produces three different files, consistent with the the Google TensorFlow™ format we use internally (see Figure 1.11). The meta graph file can be thought of as storing the network architecture, the other files represent the weight settings (affected by training). However, for the options currently offered in eCognition, this does not really matter. You can simply think of all files together as representing the model.



Name	Type	Size
model.data-00000-of-00001	DATA-00000-OF-0...	88 KB
model.index	INDEX File	1 KB
model.meta	META File	11 KB

Figure 1.11: *Model files exported by algorithm save convolutional neural network.*

If you train your model for a very long time, saving out intermediate states is always a good practice. In some cases, you may find that during learning your model suddenly produces NaN values on output. (In that case, you probably want to lower your learning rate.) Also, if you train your model for too long — perhaps showing it the same samples over and over again — you may run into overtraining. Model performance may still further increase on your training set, but it may get worse on your test data.

Lesson 2 - From class heatmap to accuracy assessment

2.0 Lesson content

In this lesson, we provide you with a prototypical rule set of how to use a class heatmap for detecting actual targets. We will show you how to choose an optimal threshold, and how to estimate different types of error rates. This provides you with important quantitative information on the quality of your trained network, and allows you to compare it to other approaches such as template matching (also available within eCognition Developer).

You will learn to:

1. **Smooth the heatmap**, to get a good local estimate of target presence.
2. **Detect local maximal exceeding a certain threshold** in the smoothed heatmap.
3. Select an **optimal threshold**, that minimizes number of errors.
4. Create a **vector layer of target locations**.
5. Evaluate different types of **errors and error rates**.
6. **Compare the obtained classification accuracy** to that achieved by a template matching approach.

2.1 Heatmap smoothing

Target presence is usually indicated by several pixels of high values around the target location. Therefore, a smoothed average is expected to more accurately indicate target presence than individual pixel values.

- **execute the process SMOOTH HEATMAP** to generate and display the layer *smoothHeatmap* (see Fig. 2c)

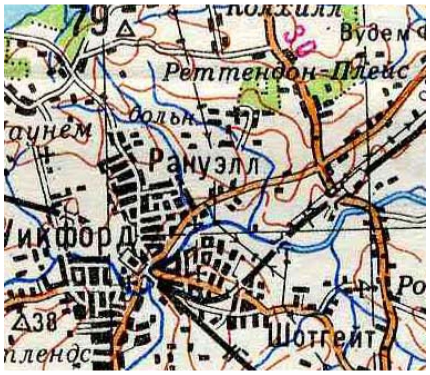


Figure 2.1a: *The original image*

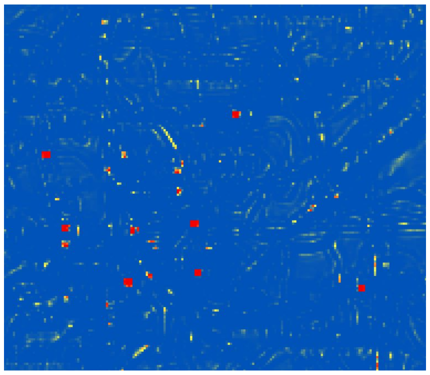


Figure 2.1b: *Layer TargetHeatmap*

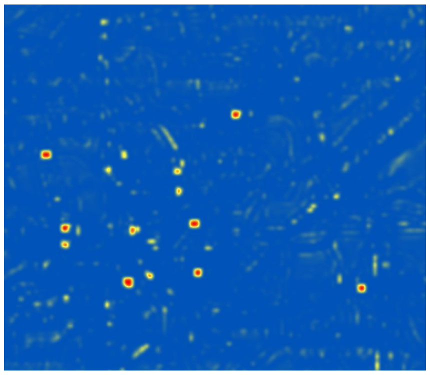


Figure 2.1c: *Layer smoothHeatmap*

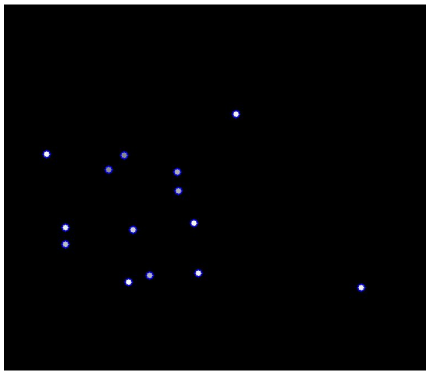


Figure 2.1d: *Layer TargetLoc and Targets*

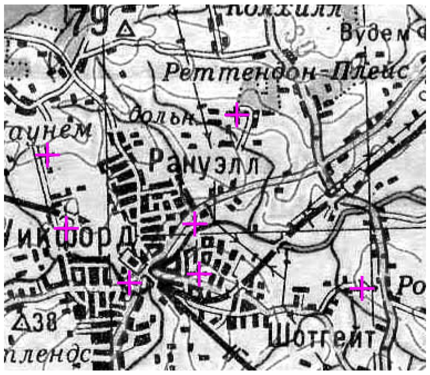


Figure 2.1e: *Targets found with optimal threshold*

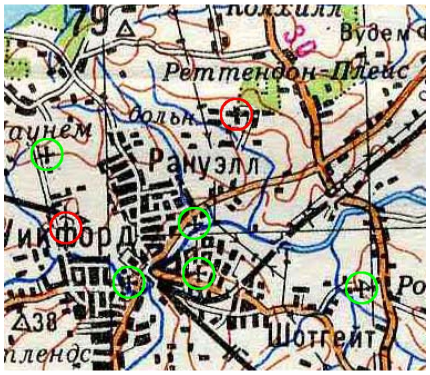


Figure 2.1f: *Visualization of false and correct detections*

2.2 Find local maxima

Target locations are expected to coincide with local maxima (and high values) in the heatmap. With “local” we have a specific distance in mind, as targets cannot appear just a few pixels from each other.

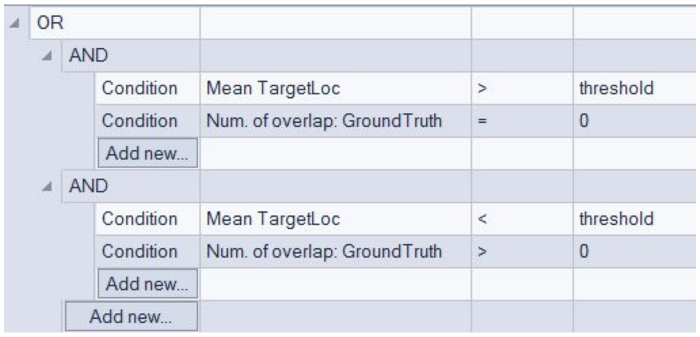
- **execute the process *pixel filter 2D morphology(dilate)*** to create a layer *localMax*. The localMax layer reflects the maximum value of the smoothedHeatmap in a neighborhood of 9 pixels (the parameter *iterations* defines the radius). Thus, when *localMax* and *smoothHeatmap* have the exact same value, we are at a local maximum of the *smoothHeatmap* layer.

We start out by finding far too many targets, applying a rather liberal threshold of 0.5 (Later, we will drop targets based on an optimized threshold.).

- **execute the *update variable* process** to set the threshold variable to 0.5.
- **execute the *layer arithmetics* process** to generate the layer *TargetLoc.* This process first describes two conditions for target presence: the pixel has to be a local maximum of the SmoothedHeatmap layer (localMax=SmoothedHeatmap) and it needs to exceed the threshold (SmoothedHeatmap>threshold). For pixels for which both conditions are true, this will evaluate to 1. We then multiply with the value of SmoothedHeatmap. The value of the *TargetLoc* layer is thus evaluates to the value of the SmoothedHeatmap for local maxima above 0.5. All other pixels will have the value 0.
- **execute *pixel filter 2D: morphology(dilate)*** to enlarge the extent of the TargetLocs in the layer.
- **execute the *multi-threshold segmentation*** to create circular Target objects. Note that our local maximum constraint ensures that there are no overlapping targets, nor targets that touch each other.
- **execute set *custom view settings*** to show the *TargetLoc* layer and the *Target* object outlines (see Figure 2.1d).

2.3 Select optimal threshold

Next we select an optimal threshold, in the sense that it minimizes the number of errors. We could also set a fixed threshold, which may be more reasonable in a real life scenario (in particular if we have no ground truth available, on which we base the optimization). However, here we are interested in getting a feeling for the quality of our model, and it seems fair to compare models based on the minimal number of errors that can be achieved.



OR			
AND			
Condition	Mean TargetLoc	>	threshold
Condition	Num. of overlap: GroundTruth	=	0
Add new...			
AND			
Condition	Mean TargetLoc	<	threshold
Condition	Num. of overlap: GroundTruth	>	0
Add new...			
Add new...			

Figure 2.2: *Conditions for selecting target objects reflecting errors.*

- **execute the compute statistical value operation** to estimate the number of errors given for the current threshold setting (0.5). Note that the first AND condition finds false detections (the value is above our criterion, but there is no ground truth entry). The second AND condition finds missed targets (the value in TargetLoc is not high enough to be found as a target, but there is a corresponding ground truth entry).

Any object meeting either the first AND condition or the second AND condition are counted as errors (see Figure 2.2).

- **execute the update variable process** to initialize the currentErrorMin estimate
- **execute the update array proces**s to clear the array *optimalThresholds* in which all threshold values will be stored which produce this currentErrorMin
- **execute the loop: while threshold < 1**, which will increase the threshold step by step, and re-evalute the number of errors that would be obtained with this threshold. If a new minimum is achieved, the currentErrorMin is updated and any old values in optimal are discarded. If the number of errors obtained with the current threshold correspond to the currentErrorMin, the threshold is stored in the array *optimalThresholds.*
- **execute the select value close to median of all optimal thresholds**, which will set the optimal threshold to an intermediate value of all the optimal threshold values found.

2.4 Create target vector layer

- **Execute the process CREATE TARGET VECTOR LAYER** to generate and display a vector layer *Targets* containing all *Targets* that are found using the optimal threshold identified previously.

2.5 Evaluate error rates

Next we want to compare the vector layers *Target* and *GroundTruth*, and quantitatively evaluate the quality of our convolutional neural network.

- **Set a breakpoint** on the process *on TestMap* inside the customized algorithm *evaluateErrorRates,* and **execute the process COMPUTE ERROR RATES**.
- Go to the **CustomizedAlgorithm tab** in the process tree window, **remove the breakpoint**, and **execute the processes classify targets, ground truth, and tolerance zones for correct detection** to classify circular regions around the ground truth as class *Tolerance.* Pixels corresponding to the ground truth are classified as class *GroundTruth*, and pixels corresponding to the vector layer Target as class *Target* (see Figure 2.3). Be aware that in this rule set *GroundTruth* objects will be reclassified as Target when the position of Target and Ground Truth overlap exactly. This is not a problem, as we will only use class *GroundTruth* to identify Misses, and in that case, *GroundTruth* is not overwritten quasi per definition by class *Target*.

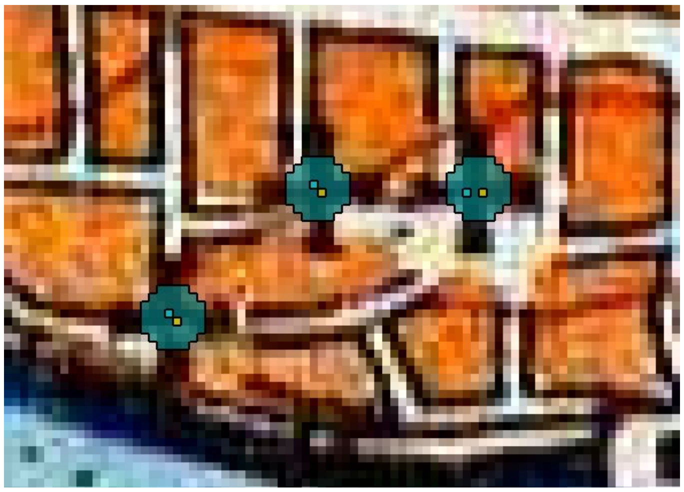


Figure 2.3: *Tolerance regions (petrol) around pixel size objects of classes GroundTruth (yellow) and Target (cyan).*

- **execute subclassify tolerances** to obtain the following classes:
    - *Tolerance_Miss* does not contain a *Target* object,
    - *Tolerance_SingleHit* contains exactly one *Target* object, and
    - *Tolerance_MultiHit* contains several *Target objects.* Remember that we required that *Targets* reflect local maxima in a raster layer, therefore, different *Target* objects cannot be very close to each other, and we do not expect any *Tolerance_MultiHit* objects here. They are thus not considered further in this tutorial, but you may have to do so if you adapt the ruleset for your own use case. (You would have to make sure that only one *Target* in the *Tolerance_MultiHit* object is counted as *Hit*, and the others as *False*.)
- **execute the process identify hit, miss, and false** to classify pixel-sized objects of classes *Hit*, *Miss*, and *False*. Note that objects close to the scene border, where results are not fully valid due to lack of context information, are removed.
- **execute create vector layers** to create (temporary) point vector layers and buffered vector layers for Hits, Misses and Falses, and to visualize correct detections (in green), false detections (in red), and missed targets (in yellow).
- **execute errors and error rates** to compute various accuracy estimates

2.6 Comparison to template matching

- **execute process standard template matching** to generate a template based on the Ground Truth in the training region (map main) and apply it on the TestRegion. Even though the threshold in the template matching algorithm is optimized **41 errors occur in the test region**.
- **execute process template matching with mask** to generate a template with a mask (this takes a few minutes) and apply it on the test region. For this more advanced template (where pixels in less informative regions of a 19x19 pixel patch are ignored) the **number of errors is reduced to 22**.

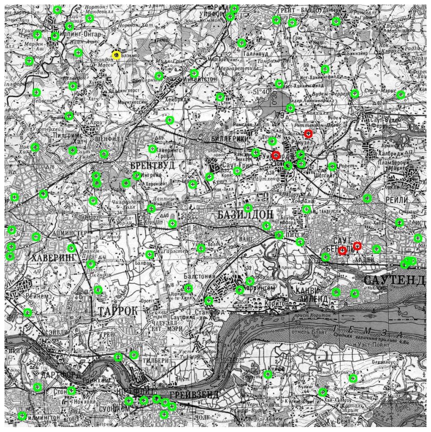


Figure 2.4a: A *typical result obtained with the tutorial. The 90 hits are shown in green, the one missed target is shown in yellow, and the 4 false positives are shown in red.*

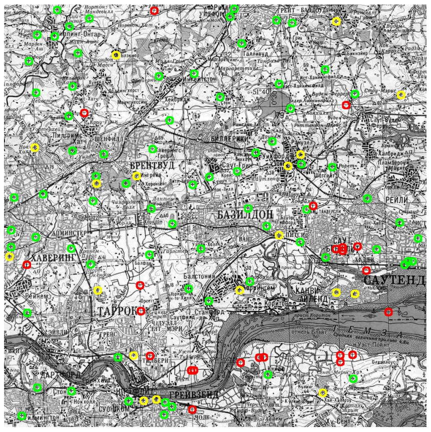


Figure 2.4b: *Optimal results with the standard template matching approach (41 errors, with 18 missed targets and 23 false positives).*

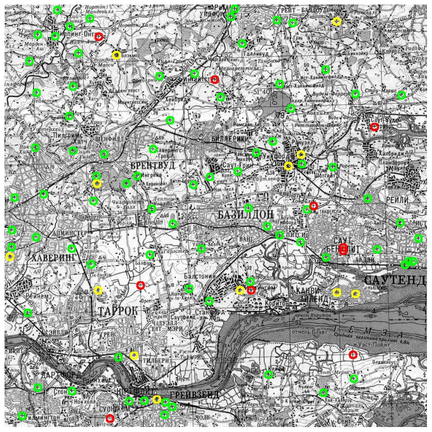


Figure 2.4c: *Optimal results with the masked template matching approach (22 errors, with 12 missed targets and 10 false positives).*

The convolutional neural network created in this tutorial will not always give the same number of errors, as there are random processes at work, e.g., when samples are generated in the sample folder, and when specific samples are selected for each training batch. Running the tutorial 253 times, we obtained the distribution of error numbers shown in Figure 2.5.

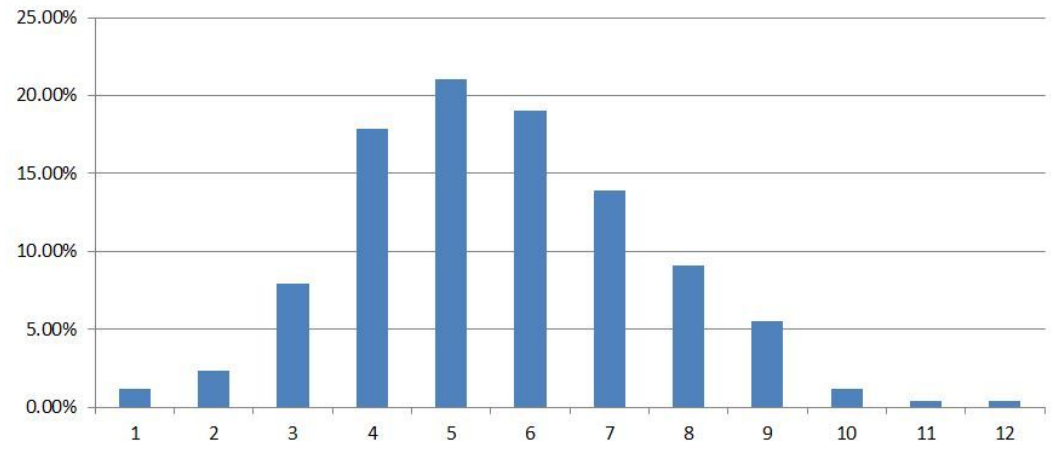


Figure 2.5: *Relative frequency of different results for the number of errors based on 253 runs.*

This means that the number of errors is on average around 5.6 errors, representing more than a **7-fold improvement over standard template matching**, and an approximately **4-fold improvement compared to template matching with a masked template**.

We believe this illustrates that even very simple convolutional neural networks can be useful in practical applications.

Where to get additional help & information?

The eCognition Community

The eCognition Community helps to share knowledge and information within the user, partner, academic and developer community to benefit from each other's experience.

The Community contains content such as:

- **Wiki**: collection of eCognition related articles (e.g. Rule Set tips and tricks, strategies, algorithm documentation...).
- **Discussions**: ask questions and get answers.
- **File exchange**: share any type of eCognition related code such as Rule Sets, Action Libraries, plug-ins...
- **Blogs**: read and write insights about what's happening around our industry…

Share your knowledge and questions with other users interested in using and developing image intelligence applications for Earth Sciences at:

http://community.ecognition.com/.

The User Guide & Reference Book

Together with the software a User Guide and a Reference book is installed. You can access them in the Developer interface in the main menu 'Help>eCognition Developer User Guide' or Reference Book.

The Reference Book lists detailed information about algorithms and features, and provides general reference information.

eCognition Training

eCognition Training Services offer a carefully planned curriculum that provides hands-on, real-world exercises. We are dedicated to enhancing customers' image analysis skills and helping these organizations to accomplish their goals.

Our courses are held in our classrooms around the world and on-site in our customer's facilities. We offer regular Open Training courses, where anyone can register and In-Company Training. We also offer Customized Courses to meet a customer's unique image analysis needs, thereby maximizing the training effect.

For more information please see our website or contact us at: eCognition_Training@trimble.com