Abstract

This report gives an overview and motivates the design of a C++ framework for object recognition using channel-coded feature maps. The code was produced in connection to the work on my PhD thesis *Channel-Coded Feature Maps for Object Recognition and Machine Learning*. The package contains algorithms ranging from basic image processing routines to specific complex algorithms for creating channel-coded feature maps through piecewise polynomials. Much emphasis has been put in creating a flexible framework using virtual interfaces. This makes it easy e.g. to switch between different image primitives detectors or learning methods in an object recognizer.

Some common design choices include an image class with a convenient but fast pixel access, a configurable assert macro for error handling and a common base class for object ownership management. The main computer vision algorithms are channel-coded feature maps (CCFMs) including their derivatives, single-sided colored lines, object detection using an abstract hypothesize-verify framework and tracking and pose estimation using locally weighted regression and CCFMs.

The code is considered as having alpha status at best. It is available under the GNU General Public License (GPL) and is mainly intended for future research on the subject.
1 Introduction

This document describes parts of the C++ code produced in connection to my PhD thesis Channel-Coded Feature Maps for Object Recognition and Machine Learning [2]. The code is available under the GNU General Public License (GPL) and is mainly intended for further research on the topic. A reference for the entire package is available as automatically generated HTML pages by Doxygen [1]. This report complements this reference with a more detailed discussion and motivation for different design choices. While the HTML documentation is most likely enough for a user, this report is intended for anyone who wants to understand or extend the code. I strongly recommend having access to the HTML reference while reading this document.

1.1 Overview

The package contains routines for:

- Basic image processing, consisting mainly of a convenient Image class and some edge filtering routines.
- Channel-Coded Feature Maps (CCFM) including derivatives, using the piecewise polynomial approach and various preprocessing modes.
- Single-Sided Colored Lines (SSCL) (line segment primitives)
- Locally Weighted Regression (LWR) with adaptive kernel width.
- Tracking, view based pose estimation and the combination of both.
- A complete object recognition framework, with virtual interfaces for the similarity detector, hypothesizer and verifier
- Various demos and MEX interfaces to selected parts of the code.

1.2 Disclaimer

The code can be considered as having alpha status at best. I tried to make the code as tidy and well-documented as possible given the time constraints, but there are probably still many glitches. The main purpose of releasing the code under GPL is to give free access to it by members of the Computer Vision Lab at Linköping University and their collaborators. The code is available by request for anyone, but bear in mind that it is research code under development, and not a complete framework in any way.
2 Common Design Choices

2.1 Naming Conventions

The following conventions have been adopted

- Abstract base classes are prefixed with capital `I`.
- Class member variables are prefixed with `m_`, except for public variables in small struct-like classes.
- Function and variable names start with a lowercase letter while class and type names start with uppercase.

2.2 Copying and Wrapping Functions

In several cases, I submit a non-const reference to a variable as an “output parameter” instead of letting a function return a value, as in:

```cpp
template<class T>
void copyTo(const mxArray* source, CvlImage<T>& dest);
```

There are two main reasons for this. The first reason is that this design lets the caller decide about the lifetime and scope of the output. Some examples include

```cpp
// Create an image in the local scope
CvlImage<double> img;
copyTo(myMxArray, img);

// Create an image on the heap
CvlImage<double>* img = new CvlImage<double>;
copyTo(myMxArray, *img);

// ..or directly as a class member variable
copyTo(myMxArray, m_memberImage);
```

If the `CvlImage` was returned the traditional way, it would either be as a pointer to a heap-allocated object or by value, in which case we risk copying all image data a second time if the compiler fails to optimize away this copying.

The second reason is that function overloading based on return type only is not allowed. We would need different copying functions like `copyToMxArray`, `copyToCvlImage` etc. By disguising the return value as an input parameter, we can have several functions with the same name:

```cpp
template<class T>
void copyTo(const mxArray* source, CvlImage<T>& dest);

template<class T>
void copyTo(const mxArray* source, vector<T>& dest);
```

Having fewer names to remember reduces code complexity.

In order to keep the design uniform, the opposite `CopyTo` functions work in the same way:
Here, the perhaps confusing reference to pointer construction (**) is used. This means that the actual pointer submitted to copyTo can (and will) be overwritten.

The following code is perfectly valid:

```c
mxArray* mx = 0;
copyTo(myCvlImage, mx);
// Now mx points to a recently allocated mxArray
```

In contrast, the following code would produce a memory leak:

```c
mxArray* mx = mxCreateDoubleMatrix(100, 100, mxREAL);
copyTo(myCvlImage, mx); // Warning: Loses the previously created mxArray
```

### 2.3 Error Handling

Exceptions are not used. The error handling is implemented in a very simple but convenient way in the files `core/debug.h` and `core/debug.cpp`. The macro `myAssert` takes a boolean condition and an error message and prints an error message if the condition fails. The macro can be configured to break execution using `exit` or `mexErrMsgTxt` depending on whether a Matlab MEX file is being compiled.

### 2.4 Memory Management

Sometimes when an object obtains references to other objects, it is more convenient to let the object also obtains ownership (i.e. delete responsibility) of these references. Consider the following toy code:

```c
HomeTheater* createDvdPlayer(){
    DvdPlayer* dvd = new DvdPlayer();
    TvSet* tv = new TvSet();

    HomeTheater* theater = new HomeTheater();
    theater->setDvdPlayer(dvd);
    theater->setTvSet(tv);

    tv->setOwnerTag(theater);
    dvd->setOwnerTag(theater);
}
```

Here, we need the `HomeTheater` to obtain ownership of the `TvSet` and the `DvdPlayer`, otherwise no one would be responsible for deleting these objects. In other situations however, an ownership transfer may not be desired. Making the transfer explicit by `setOwnerTag` rather than implicit by the `setDvdPlayer` and `setTvSet` increases flexibility and makes it more obvious what is going on.

The `setOwnerTag` method is defined in a base class `OwnerTag`, which also contains a void pointer to the owner of the object. There is also a global function `deleteIfOwner` that lets an object delete a pointer only if it owns it. This helps resolving ownership issues. The destructor of `HomeTheater` would look like this:
Using the OwnerTag design, anyone handing out pointers to a borrower also has the option of making the borrower owner of the object. As a consequence, every object that keeps references to other OwnerTag-objects might unknowingly become owners of the objects and should call deleteIfOwner on these pointers in their destructors.

2.5 Enabling Efficient GPU Processing

The graphics processor is especially well-suited for image processing, and can provide a large increase in efficiency for many common vision tasks. In this implementation, the GPU is used only in the OpenGLResampler class, which cuts out resampled patches from a query image given a position, scale and orientation parameter.

Even though the GPU is fast, transferring data between the GPU and CPU memory is relatively slow. If we are to cut out 15 patches from a query image, it is best to upload the image once, cut out all patches, and download the result once. This would not be possible using a straight-forward type signature like

```cpp
void resample(const CvImage<uchar>& source,
              const SimilarityFrame& frame,
              CvImage<uchar>& dest);
```

From a GPU perspective, it would be better to accept a source image, a list of similarity frames, and return a list of patches. However, consider a hypothesize-verify approach for object recognition. Here, we might need to call the resample function once for each hypothesis. To avoid having to upload the query image to GPU memory more often than necessary, I propose having one function for setting the query image and another function for running the actual method. This design then spreads to other parts of the system, giving e.g. the partial interface to a tracking function as

```cpp
void setSourceImage(const CvImage<uchar>& source,
                     const CvImage<uchar>* mask);

void refine(vector<SimPoseHyp>& hyps);
```

The function setSourceImage can then upload an image to the GPU, and refine can run the actual processing and download a chunk of results.

The GPU is far from fully utilized in the implemented system, but at least the system is designed to enable optimizing a specific module using the GPU without having to change the global system structure too much.

2.6 Matlab Interfaces

Most basic types like images and vectors can be copied to and from mxArrays using the copyTo functions described in 2.2 and defined in e.g. core/CvImage_Matlab.h
and core/MatlabTools.h. This makes the actual mexFunction very compact (see imgproc/ColorEdges_mex.cpp for an example).

In some cases, a MEX function needs to store its state between subsequent calls. Consider the LwrInterpolator class, which implements the locally weighted regression method and offers a method newSample for training and run for operation. A MEX wrapper for this class must make itself persistent in memory, instantiate an LwrInterpolator object and support calling both member functions newSample and run. Furthermore, it may be necessary to instantiate not only one but several LwrInterpolator objects for different learning problems. A Pseudo-code MATLAB usage example for LwrInterpolator_mex is

```matlab
handle = LwrInterpolator_mex('init', inputSpaceSize, outputSpaceSize);
for ii = 1:10
    [inputSample, outputSample] = getTrainingExample();
    LwrInterpolator_mex('newSample', handle, inputSample, outputSample);
end
answer = LwrInterpolator_mex('run', handle, getQueryPoint());
LwrInterpolator_mex('clear');
```

The MEX file is able to create several instances of the LwrInterpolator object and return a handle to each one. It is then possible to call the functions newSample and run using a handle to one of these previously created instances.

This is realized using the TMexObjectList class template in core/TMexObjectList.hpp. This class stores pointers to dynamically allocated objects of a template type T and wraps references to these objects as mxArray’s. The usage is probably best explained by an example:

```cpp
// ---- Globally in the MEX file ----
// (hence the prefix 'g')
// Should be persistent between calls to the mex
TMexObjectList<LwrInterpolator> g_lwrInterpolators;

// ---- In "mexFunction" ---

// If "init", create a new LwrInterpolator and return a handle to it.
// Also lock the mex file such that it is not cleared by MATLAB’s
// memory manager.
mxArray* handle = g_lwrInterpolators.giveNew(new LwrInterpolator());
plhs[0] = handle;
mexLock();

// If "newSample" or "run", access a given LwrInterpolator instance
// based on an mxArray handle
LwrInterpolator* lwr = g_lwrInterpolators.get(prhs[1]);

// If "clear", clear all instances and unlock the MEX file such that it
// can be unloaded from memory
g_lwrInterpolators.clearAll();
mexUnlock();
```

For a real usage example, see e.g. learning/LwrInterpolator_mex.cpp. One nice thing about TMexObjectList is that objects are accessed using mxArray handles, but it is not visible to the user how these handles are created or what they contain. In the current implementation, the handles are only running integers, but this could
(and should) be changed to something safer.

3 Core Classes

3.1 CvIImage

The CvIImage class is designed as a low-weight image class, with simple access to dimensions, individual pixels, and the raw data pointer. The pixels can be accessed via overloaded ()-operators, using either linear indices or (x,y)-coordinates. These operators return references to pixels, and can be used both to set and get values in the spirit of the []-operators for STL containers. The () operators were chosen since [] does not support multiple indices. A simple usage example:

```cpp
// 640x480 single-channel image
CvlImage<uint8_t> image(640, 480);
image(10, 20) = 5; // (x,y)-indexing
image(1023) = 4; // linear indexing
cout << image(10, 20);
cout << image(1023);

// 100x200 RGB image
CvlImage<uint8_t> image(100, 200, 3);
image(10, 20, 0) = 5; // (x,y,color)-indexing
image(1234) = 4; // linear indexing
cout << image(10, 20, 1);
cout << image(1234);
```

By default, these indexing operators perform bound checking and aborts with myAssert when an index is out of bounds. For maximum speed, this check can be turned off by defining CVLIMAGE_REMOVE_BOUNDS_CHECK to the preprocessor. When this is done, the linear indexing is as fast as working on raw pointers directly.

3.2 Geometry

There are a number of classes for representing geometrical primitives (Line, Ellipsoid, Point, DblVect2D, SimilarityFrame). All angles are measured in radians. A SimilarityFrame defines a local limited coordinate system (a box) in an image using a position (x,y), rotation \( \alpha \) and scale \( s \). The scale parameter is the orthogonal distance from the center to the boundary of the box. The rotation is a counterclockwise rotation of the frame in the image, or (equivalently) a clockwise rotation of the image in the patch.

The image coordinate system is defined with the x-axis pointing right and the y-axis pointing down, and with (0,0) in the upper-left corner. Often, the local coordinate system within a similarity frame is denoted with \((u,v)\) while the image coordinate system is denoted with \((x,y)\). Also in this local coordinate system, the y-axis points downwards. This is in contrast to the mathematical description in the thesis, which uses the common mathematical convention of having the y-axis pointing upwards. This causes a slight difference between some geometrical expressions in the code and the thesis.
A SimilarityTransform can be constructed from two lines or similarity frames, and is internally represented using matrices \( \mathbf{A} \), \( \mathbf{b} \) and the image of a canonical frame. The \( \mathbf{A}, \mathbf{b} \) define the transform according to

\[
x \leftarrow \mathbf{A}x + \mathbf{b} = \begin{bmatrix} a_0 & a_2 \\ a_1 & a_3 \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} + \begin{bmatrix} b_0 \\ b_1 \end{bmatrix},
\]

where the subscripts show how the matrix elements are ordered in the internal \texttt{vector<double>} representation. For convenience, the SimilarityTransform also carries along the image of a canonical frame (similarity frame where \( s = 1, \alpha = x = y = 0 \)), which simplifies some operations. The design motivation is to make the transformation functions as fast as possible at the expense of the transform construction and memory requirements, since often the same transformation is used to transform many points or patches from an object.

4 Single-Sided Colored Lines

4.1 Basic method

The Ssc1Extractor extracts line segments from edges of a color image. The basic idea is that an edge often separates an object from the background, and since each side of an edge comes from different objects, we should only look at information at one side of the edge at a time. A single-sided colored line is a linear segment of large gradient magnitude that has a homogeneous color on one side of the edge.

The basic method of computing these line segments is to start by extracting a number of seed points with large gradient magnitude, and try to grow a line segment from each of these seed points by iteratively varying the orientation and position. For each orientation and position, a line is grown in both ends until a score is maximized. This score is a combination of the gradient magnitude at the line segment pixels and the color variation at one side of the edge.

4.2 Implementing ISimilarityDetector

In order to make it easy to switch between different image primitives in the system, each primitives detector should implement the ISimilarityDetector interface:

```cpp
class ISimilarityDetector
{
public:
    virtual void run(const CvlImage<uint8_t> & img,
                      const CvlImage<uint8_t> * mask,
                      vector<SimilarityFrame> & result) const = 0;

    virtual double match(const SimilarityFrame & sf1,
                          const SimilarityFrame & sf2,
                          bool useExtraOnly) const = 0;

    virtual void setReferenceFrame(const SimilarityFrame & frame) = 0;
};
```
The reason for including the `match` function in this interface is that only the actual detector is expected to know how two similarity frames should be matched. For example, the orientation of a line feature is usually much more accurate than the scale, since the endpoints of a line are often relatively unstable.

A problem here is that when verifying an object hypothesis by matching primitives, the best approach may not be to compare similarity frames independently from the size of the object. Consider an object with a rough size of 200 pixels. If a certain line feature of the object is 5 pixels away from its expected position, it should be given the same cost regardless of whether the line length is 10 or 100 pixels. In other words, it is better to compute the cost of a position error of a line feature relative to the object size rather than to the line size. This is where the function `setReferenceFrame` comes in. This function can be used by an object hypothesis verifier to select the frame in which the entire object is believed to be located. Then, patches are matched by calling `match`, which now takes the reference frame into account when computing the costs.

5 Channel-Coded Feature Maps

The creation of channel-coded feature maps is distributed over three main classes. The `encoder` creates the actual encoding from a multi-channel feature image. The `preprocessor` creates a feature image from an RGB image, and the `extractor` cuts out a patch from a similarity frame in a larger image and runs both the preprocessing and encoding. A typical user will primarily work with the class `CcfmExtractor`.

5.1 The Encoder

The input to the actual `encoder` is a multi-channel image where each channel contains one input feature. As an example, if local orientation, hue and saturation are to be encoded, the input should be an $M \times N \times 3$ double image with values in $[0, 1]$.

The abstract base class `IViewEncoder` in `core` defines the functions

```cpp
virtual void encode(
    const CvIImage<double>& image,
    const CvIImage<double>* weights,
    FeatureVector& result);

virtual void encodeWithSimilDerivs(
    const CvIImage<double>& image,
    const CvIImage<double>* weights,
    FvWithSimilDerivs& ccfm);
```

This is intended for any class that creates a fixed-length feature vector from a preprocessed patch and can be used to seamlessly switch between e.g. channel-coded feature maps, DCT or PCA representations etc. The `ICcfmEncoder` completes this interface with an additional function

```cpp
virtual void setup(
    ChEncoding finalEncoding,
    bool needsDerivatives,
    const vector<int>& nChannels,
```
const vector<ChPlacement>& placement);

This defines the parameters of the desired CCFM encoding. This interface can be used to seamlessly switch between different implementations / algorithms of CCFMs, e.g. the piecewise method in \texttt{PwCcfmEncoder} and a more direct version.

The actual implementation of the piecewise encoding is relatively complex, and an overview of this implementation is given in a separate section. There is also an older implementation of a direct encoding available in \texttt{XygDiffEncoder}, but it is limited to a single image feature.

5.2 CCFM Memory Layout

The first two channel dimensions are always the spatial coordinates, starting with \(y\) (to maintain compatibility with some older code). The channel grid is organized such that the least significant index is the first dimension. For a 5D encoding (spatial coordinates and three features), the index of a given channel \((y, x, f_1, f_2, f_3)\) is

\[
  k = y + n_y(x + n_x(f_1 + (n_{f_1}(f_2 + (n_{f_2}f_3)))))
\]

For the monopiece encodings used as a first step of the piecewise encoding, the monopiece index is the least significant index. In this case, the linear index \(k\) of a certain monopiece \(p\) at a certain position \((y, x, f_1, f_2, f_3)\) in the channel grid is computed as

\[
  k = p + n_p(y + n_y(x + n_x(f_1 + (n_{f_1}(f_2 + (n_{f_2}f_3)))))
\]

This is in accordance to previous P-channel encodings by Michael Felsberg and Johan Hedborg.

5.3 Implementation of the Piecewise Encoding

A detailed description of the piecewise polynomial encoding algorithm is given in Chapter 4 of my thesis [2], and the code is found in \texttt{ccfm/piecewise}. This section describes the implementation and assumes that the reader is familiar with the algorithm. The implementation is rather complex, and this section is only recommended to a reader that really wants to dig into the piecewise CCFM code.

The first step of the piecewise encoding is the monopiece encoding. This is created by the function template \texttt{enc\_general} in \texttt{pwpolyenc\_main.cpp}. The template parameter is a function computing the values of all required monomials at a given bin. The \texttt{enc\_general} function loops through the image and for each pixel computes the bin index and coordinate offsets within the bin. It then calls the template parameter function, which computes the values of all monomials from these offsets. Since different target encodings require different monomials, this designs makes it possible to switch between different monopiece encodings while reusing the rather complex index and offset computations. The template solution was chosen for speed to avoid the virtual function call overhead for a subclassing method. Some inner
monomial computation functions are available in `pwpolyenc_inner.h` (placed in a header file to facilitate inlining).

The second step of the method is to perform a series of multi-dimensional convolutions. This is done by `PwCcfmEncoder` with the help of `NdConvolver` written by Johan Hedborg. The `PolynomialOrderMap` class represents the monomials used in the monopiece encoding. For example, if `polyOrderMap.get(5, 1)` returns 2, this means that monopiece number 5 has order 2 in dimension 1 (second spatial dimension). For each inner encoding function in `pwpolyenc_inner.h`, there is a corresponding polynomial order map defining the same set of polynomials. In principle, the polynomial order map could be used to do the actual inner encoding, but I found no way to make this as fast as the current solution.

In order to help avoiding mistakes and make the code easier to understand, some special index types were introduced. This reduces the risk of mixing up indices, since the compiler complains when one index type is used where another is expected. These types are defined in `CcfmTypes.h` and should be as fast as integers in release mode. The type `PolyOrderIx` is used to represent the order of a polynomial in one dimension. The `DimIx` is used to represent a dimension \((y, x, f_1, f_2, \ldots)\), and the `MultidimPolyIx` indexes the multi-dimensional monopieces produced by the monopiece encoder.

As an option, `USE_NON_STRICT_INDEX_TYPES` can be defined to the compiler. This defines the types as raw integers, which might resolve potential compilation issues and ensures maximum speed but removes some compiler checks.

6 Learning

6.1 Classification

A simple nearest neighbor classifier `NnClassifier` is available in `learning`. This class implements the `IClassifier` interface in order to enable switching between different classifiers in a system. The `NnClassifier` can be configured with different distance measures, which are implemented by different subclasses to the `IDistance` template. There is a slight overhead for the virtual function call, but since this class is intended for use on CCFMs, the cost of computing a distance between two feature vectors is large enough to make the function call overhead negligible.

6.2 Locally Weighted Regression

A basic version of the locally weighted regression algorithm is implemented in `LwrInterpolator`. The scale selection strategy is factored out of the main class in order to enable switching between strategies (see `ScaleSelectionStrategies.cpp`). Furthermore, any basis function implementing `IMapping` can be used. Only the constant and linear bases are supported currently. The linear basis is recommended, and I see little room for improvement by introducing a higher-order basis.
6.3 Inverse Iterations

If we want to learn a mapping from a high-dimensional to a low-dimensional space (e.g. feature vectors to pose parameters), which is essentially an inverse problem, it is suggested in Chapter 10 of my thesis to instead learn the more direct forward model (pose parameters to feature vectors) and then invert this mapping using an iterative optimization procedure. This is implemented by \texttt{NnWithInverseInterpolator}.

This class keeps a reference to an \texttt{NnClassifier}, used to find the initial guess, and to a general \texttt{ILearnedDerivativeMapping} for representing the forward model. The example in the thesis is to use an \texttt{LwrInterpolator} as forward model, but any trainable mapping that computes derivatives can be used.

7 Tracking and Recognition

7.1 Tracking

The tracking and pose estimation using Gauss-Newton optimization is split into several classes. The \texttt{GaussNewtonOptimizer} in \texttt{math} contains the actual optimization procedure and is configurable with any target function that implements the \texttt{IDerivativeMapping} interface (defined in \texttt{AbstractMapping.h}). This makes it possible to use the same core optimization procedure both for the tracking, pose estimation and the simultaneous optimization of both.

The \texttt{SimTracker} in \texttt{tracking} tracks a similarity frame based on features with derivatives obtained by any \texttt{ISimPatchExtractor}, e.g. the \texttt{CcfmExtractor}. In order to use the general \texttt{GaussNewtonOptimizer}, there is a small wrapper class that exposes the \texttt{ISimPatchExtractor} as an \texttt{IDerivativeMapping}. For an example of tracking a similarity frame using channel-coded feature maps, see \texttt{CcfmTrackingDemo} in \texttt{demos}.

7.2 Simultaneous Tracking and Pose Estimation

The \texttt{SpetTracker} (Simultaneous Pose Estimation and Tracking) in \texttt{tracking} combines an object model and a feature extractor and optimizes all parameters simultaneously. The object model can be any \texttt{IDerivativeMapping} giving the expected feature vector description of a view as a function of a set of pose parameters (e.g. pose angles). The feature extractor can be any \texttt{ISimPatchExtractor} extracting a feature vector from a query image according to a set of similarity parameters. The \texttt{SpetTracker} defines a wrapper that combines the outputs of the feature extractor

\footnote{It is only available in an older Matlab implementation available in the LiU/CVL-Matlab repository \texttt{pattern\_recog/LWR} (restricted access for external users).}
and the view model into a single target function which is optimized by the general GaussNewtonOptimizer.

There is also a specialization of this class called LwrCcfmSpetTracker which uses a CcfmExtractor as view feature extractor and an LwrInterpolator as object model. This is the procedure described in the thesis. However, using virtual interfaces, it is possible to use the SpetTracker together with another feature extractor or object model.

7.3 Object Recognition

A basic object recognition framework is available in objrec. A hypothesize-and-verify approach is used, where the HypAndVerifier class runs the main hyp-verify loop, and the actual hypothesizer and verifier can be varied using virtual interfaces. Currently, the only choices are PatchBasedHypothesizer, PatchBasedObjrecVerifier and FullViewVerifier. The patch-based approach is based on cutting out patches around image primitives. From these patches, some feature vectors are constructed and matched between the query and training views. The method is described in more detail in the thesis.

Using the interfaces ISimilarityDetector and ISimPatchExtractor, the actual primitives and view representation used can be varied freely. In Sect. 8, one specific choice is described, namely the one used in the demonstrator system of the COSPAL project.

8 The COSPAL Object Recognizer

The COSPAL project was an EU-funded project on cognitive systems running between 2004 and 2007. As part of the final demonstrator, a module for recognizing objects using channel-coded feature maps was produced. Most of the code in this package originates from the work within COSPAL. The current object recognition using CCFMs package was put together after the project end as an effort to clean up the code and make pieces of the system as simple to reuse as possible.

The class CospalDemonstratorObjrec contains the COSPAL demonstrator object recognizer recast in this new framework. Due to a redesign of parts of the code, this version is not identical to the recognizer actually used in the demonstrator. It is also not as rigorously tested as the demonstrator system, so there is probably many glitches and bugs. However, the main algorithmic features are the same, and this new version is more well-structured, flexible, and well-suited for further development.

8.1 Basic Method

First, a number of primitives are generated from the query image. These primitives can be equipped with a small descriptor based on information readily available in the primitive extraction. The COSPAL system used the SSCL features from Sect. 4,
where the color is available for free. Another example could be a homogeneous region, where the color and possibly some boundary descriptors are directly linked to the primitive used. This will be referred to as the intrinsic properties of the primitive.

On a second level, we can cut out a patch from the image based on the similarity frame defined by the primitive. From this patch, a richer feature vector (e.g. a CCFM) can be obtained. This means that there is in total three types of information available at each primitive: The similarity frame itself, the basic primitive property and the feature vector of the patch.

8.2 Hypothesis Generation

An object hypothesis is generated based on matching primitives using a matching cost combining all three types of information, where least weight is given to the geometric information (similarity frame). However, there is an option to also include geometrical constraints such that specifying a maximal change in scale or rotation between the query and training images. Another case is where we have a rough idea about where a certain object is and want to generate hypotheses only in the neighborhood of this input hypothesis (see Sect. 9.4).

8.3 Verification

There are two different ways of verifying hypotheses: full-view and patch-based verification. Both these verifiers implement the IVerifier<PbObjectHyp> interface, and the actual hypothesize-verify loop does not know which verifier is used.

8.4 The PatchStore

All patches from the training views are kept in a PatchStore object, and the patches from the current query view are kept in another PatchStore. This data structure makes it easy to enumerate all patches in the store or from a certain training view, or to access a certain patch based on its ID.

Internally, the patches are stored in a large vector. There is also a triple-jagged vector for fast access to a patch from its object, view and patch id. The PatchStore is responsible for maintaining consistency between these two structures.

When certain subsets of patches are requested, a lightweight collection_mirror is returned. This is an object that supports contents enumeration using STL-like begin() and end() operations, but which can be disposed of without destroying the data. This is a convenient way of returning something which looks as an STL container without handing out direct references to internal structures, and without copying any data. Since no virtual interfaces are used, we do not get the penalty of virtual function calls.

The user only needs to know that the select functions returns a TPatchList, which can be used as an STL container and disposed of when done. If the imple-
mentation of PatchStore is changed, the TPatchList might be redefined, but as long as TPatchList behaves in the same way, client code can be left unchanged.

9 Additional Tricks in the COSPAL System

This section describes some extra tricks which were implemented in order to improve the performance of the COSAPL system. Some of these tricks are fairly specific to the type of objects used in the project, and some are more general. Even if the more specific tricks could be considered as too specialized for our certain objects to be useful in a more general framework, it is important to design a real object recognizer such that this kind of tricks can be enabled without messing up the system structure too much.

9.1 Intrinsic Property Matching Filter (IPMF)

In the COSPAL system, puzzle pieces were only allowed to be red, green, blue and yellow, and holes were only allowed to be black. Thus, a certain line primitive from a piece should only be allowed to match red, green, blue or yellow lines, and a primitive from a hole should only be allowed to match black lines.

This was implemented using an extended intrinsic property matching filter attached to each primitive in training memory as the field frameExtraCost of the Patch class. This filter is a GmmVectorCost assigning a cost for matching this prototype to any other prototype based on the intrinsic property of the prototype. In COSPAL, the intrinsic primitive property was color, but this could be different for different primitive extractors.

9.2 Intrinsic Property Cluster Selection (IPCS)

This technique can be used when the intrinsic property is expected to be the same for all primitives of an object. Again, think about color as an example. In COSPAL, all objects were of a single color. Once a hypothesis is created from, say, a blue line in the query image, only blue lines should be allowed to participate in the verification. This is more restrictive than the IPMF described above, since it requires consistency between all primitives of an object.

The IPCS is enabled using the enableIpcs option in PatchBasedHypothesizer::Options. When enabled, the PatchBasedHypothesizer sets the optional field intrinsicPropertyCluster in the PbObjectHyp to the component of frameExtraCost best fitting the observed primitive. This explains why the frameExtraCost must be a GmmVectorCost rather than a more general IVectorCost<double>.

The PatchBasedObjrecVerifier makes sure that only primitives with the correct intrinsic property are included in the verification. On the other hand, the FullViewObjrecVerifier does not use primitives and can not use this information unless it somehow knows what the intrinsic property actually means. By enabling the useIpcsColorSegmentation option, the FullViewObjrecVerifier assumes that the intrinsic property is color, and uses a color segmenter to filter out
any color that does not fit to this color cluster. This is a slight violation to the principle that only the primitives detector knows about the semantics of the intrinsic primitives property, but I found no other simple way to implement this.

9.3 Training Primitives Filter

This feature is related to all the above. Again, it mainly concerns color but can in principle be generalized to whatever information the similarity detector chooses to place in frame.extra. By submitting a GmmVectorCost with a training view, only primitives with a color that matches to this GmmVectorCost are included in training. For the COSPAL objects, we can avoid including black lines in the training views of puzzle pieces or wood-colored lines in training views of holes.

This filter should be enabled if any of two previous features are enabled. In contrast to IPMF and IPCS, the training primitives filter requires more input from the supervisor - for each training view, the object recognizer must be told which colors to include and which to disregard.

9.4 Refinement using Constrained Hypothesis Generation (RUCHYG)

Assume that we know from a previous frame which objects are present, and that we only want updated estimates for our old object hypotheses. If full view verification is enabled, we could simply run the verifier to refine all current hypotheses, and the Gauss-Newton iterations will converge to the new position estimates. However, this only works if the objects are not moved too far. For large movements, it may be useful to rerun the entire procedure with hypothesis generation and verification, but constrained such that we only look for objects we already know are present, and only in regions relatively close to the old object positions. This procedure is called refinement using constrained hypothesis generation, and is enabled using PatchBasedHypothesizer::enableRuchyg. Only one previous hypothesis can be used at a time, so in order to update the position estimates of four known objects, we must run the entire hypo- and verify procedure four times, using each old object hypothesis as the ruchygHyp each time.

9.5 Patch Weight Tuning

Some patches are more informative than other. For the COSPAL objects, some patches contain only a boundary between two regions, while other patches contain a lot of structure. Ideally, more weight should be given to more informative patches.

An interesting option is to assign a weight to each patch and try to optimize all weights in order to maximize the recognition rate on some validation set. This approach was tried but not finished within the project. In order to prepare for adjusting patch weights, the CospalDemonstratorObjrec class supports adjusting the patch weights using the member function setWeights.
References
