N-light-N

Read The Friendly Manual

Mathias Seuret, Michele Alberti, and Marcus Liwicki

Document Image Voice Analysis group (Diva)
University of Fribourg
Boulevard de Perolles 90 - 1700 Fribourg - Switzerland

January 27, 2017

UNI
FR

UNIVERSITÉ DE FRIBOURG
UNIVERSITÄT FREIBURG

Abstract

This documentation wants to be a "user manual" for the *N-light-N* framework[1]. The goal is not only to introduce the framework but also to provide enough information such that one can start modifying and upgrading it after reading this document. This document is divided into five chapters.

The main purpose of Chapter 1 is to introduce into our notation and formulation. It refers to further literature for deeper introductions into the theory. Chapter 2 gives quick-start information allowing to start using the framework in an extremely short time. Chapter 3 provides an overview of the framework's architecture. Interactions among different entities are explained and the main work flow is provided. Chapter 4 explains how to write a custom XML script for the framework. Proper usage of all implemented commands is described. Finally, Chapter 6 explains how to extend the framework by creating your own script commands, layers (encoder/decoder), and autoencoders. Having read both Chapters 3 and 6 before starting to extend the framework is extremely recommended.

As the framework will evolve, this documentation should be kept up-to-date[2].

[1] Available at: https://github.com/seuretm/N-light-N

[2] The most recent version can be found at
https://github.com/seuretm/N-light-N/blob/master/N-light-N.pdf

Acknowledgement

The happy developers of *N-light-N* would like to thank for their help, collaboration, and feedback the following persons:

- Ingold, Rolf
- Chen, Kai
- Dias, Emanuele
- Fischer, Andreas
- Garz, Angelika
- Oener, Volkan
- Pastor, Joan
- Sayadi, Karim
- Simistira, Fotini
- Sittampalam, Arun
- Wei, Hao

This work was funded by the Swiss National Foundation (SNF) project HisDoc 2.0 with the grant number: 205120 150173.

Contents

Foreword

The foundations of neural networks are surprisingly old, as the first artificial neural networks date from the 1940s with the development of the McCulloch-Pitts neuron [1, 2, 3]. Since then, they have been used in many domains, including speech recognition [4], pattern recognition [5], stock market prediction [6], ... Decades of increasingly active research in this field have shown that artificial neural networks have a huge potential, and also that there is still much to learn.

Novel training methods and faster computers now allow to train and use neural networks which are much larger than it was feasible in the past. The term "deep learning" refers to the use of neural networks to learn data representation with different levels of abstraction, which can then be used for machine learning purpose.

Many libraries and deep learning frameworks are being used with increasing success in many fields. Accordingly to Bahrampour *et al.* [7], the most frequently used ones are Caffe [8], Theano [9], Torch7 [10], TensorFlow [11], and deeplearning4j[3]. These libraries allow for efficient training of deep neural networks with high parallelization and implementation on graphic cards, making them very useful, especially for image processing.

However, such optimizations have a two-fold cost for research and analysis. First, the performance optimization are obstacles for adaptability by non-experts, as high programming skills are required. Second, these optimizations decrease the users' analysis capabilities, and thus generate difficulties for a deeper understanding of the networks.

Thus, there is a gap that *N-light-N* aims to fill, as highly-adaptable framework for Convolutional Neural Networks (Cnn) allowing not only deep modifications with minimal effort, but also investigations on the inner working of the networks with ease. For this purpose, *N-light-N* follows a simplicity principle: base components that any user might some day want to modify should be kept as simple as possible, even if speed has sometimes to be sacrificed. Thus, barriers to entry for researchers are kept very low. It is

[3] http://deeplearning4j.org/

the authors' hope that N-ligne-N will lead to exciting discoveries about deep learning.

A more scientific presentation of *N-light-N* together with experimental validation and analysis is presented in [12]. This framework has also been used in published research on historical document analysis [13, 14, 15], historical document images clustering [16], feature selection [17], and for Arabic text detection in video frames [18]. Ongoing research is being done on novel methods for Autoencoder (AE) training, central multilayer features usage [19], document graph [20] edge labelling, deeper understanding of historical document image clustering, and novel kinds of neuron-like units.

Chapter 1

Theoretical Foundations

This chapter is a refresh for those who are already familiar with the topics of machine learning, autoencoders and neural networks. Shall this not be the case, you can read this as quick introduction but do not think that this short theory somehow replace a proper study of the subject. For getting a solid knowledge about deep learning, we recommend the following books. As introduction to neural networks, we recommend reading "Neural networks: a systematic introduction" [2]. A thorough historical survey of deep learning, "Deep Learning in Neural Networks: An Overview" [3], provides both information about past and current research on neural networks. Finally, "Deep Learning" [21] is a course covering most, if not all, aspects modern deep learning.

Stacked Convolutional Autoencoders (SCAEs) are based on three principles which will be described in the following sections: autoencoding, convolving and stacking.

1.1 Autoencoding

AEs are mostly used for computing feature vectors which can then be used for machine learning purpose. They are composed of two functions:

$$\vec{y} = E(\vec{x}) \quad \text{and} \quad \vec{x}' = D(\vec{y}) \quad \text{such that} \quad \vec{x} \approx \vec{x}'.$$

The idea is to take an input vector $\vec{x}$, encode it into $\vec{y}$ and then decode it again to $\vec{x}'$ expecting to have only little differences with the initial input $\vec{x}$. The encoding and decoding functions are learned during the training of the AE, where the training error is the difference between $\vec{x}$ and $\vec{x}'$.

An AE can be represented with the typical black box often used in computer

science. Figure 1.1 shows an AE taking as input a 3×3 pixels patch from a one-channel image and outputs a four-dimension vector.

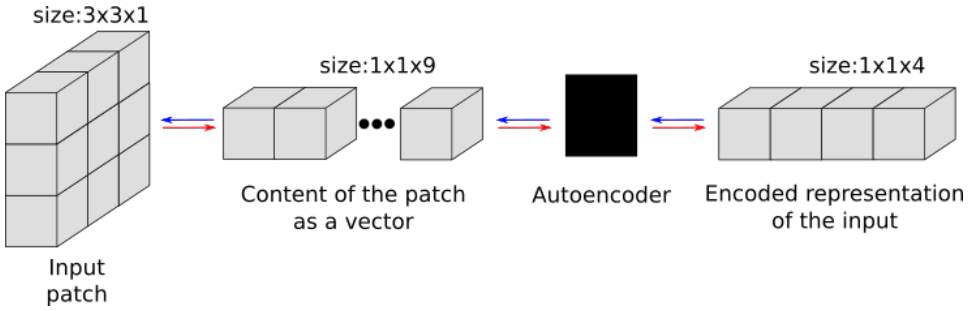

Figure 1.1: An autoencoder with his inputs and outputs. The arrows correspond to the data flow. Red arrows correspond to encoding the inputs, and blue arrows to decoding the outputs.

There are many different kinds of AE[1], and while we mostly focus on the neural network terminology for simplicity reasons, you can consider an autoencoder to be "anything that can encode and decode data".

1.1.1 Dimension of the Encoded Representation $\vec{y}$

The dimension of the encoded representation $\vec{y}$ is not fixed as it depends on the architecture of the AE itself. We have the three cases:

$|\vec{y}| < |\vec{x}|$ → the encoded output is a is a compressed representation of $\vec{x}$. This is particularly useful because if $\vec{x} \approx \vec{x}'$ holds, it means as the input can be reconstructed with a fair accuracy with less values, some unnecessary information is discarded. In other words, all information lost during dimensionality reduction are not needed and thus we have that $\vec{y}$ is the "essence" of $\vec{x}$. In computer science we often call this "essence": features.

$|\vec{y}| = |\vec{x}|$ → the encoded output is the identity $\vec{x}$. This is in general not used as the AE could just learn the identity function and would therefore not lead to useful features.

$|\vec{y}| > |\vec{x}|$ → might be done when training the AE to generate a sparse[2] representation of $\vec{x}$.

[1] For now, the software supports the following kinds of AE: autoencoding neural networks, denoising autoencoders, principal component analysis, and a naive implementation of restricted Boltzmann Machines.

[2] The implementation of a sparse `Layer` in the framework is currently under way.

1.2 Stacking

Stacking AEs literally means creating a stack with them. This means that the output of a first AE is used as input for a second (and different) AE. It can be done sequentially, typically with increasing compression rate such that $|\vec{y}_i| > |\vec{y}_{i+1}|$, where $\vec{y}_i$ is the output of the i-th layer of the stack, as illustrated in Figure 1.2.

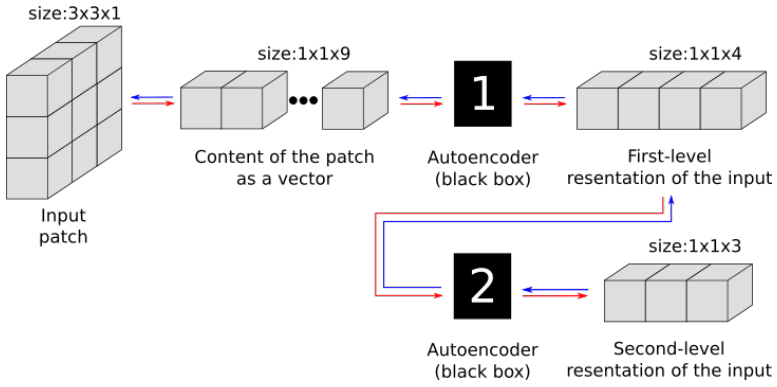

Figure 1.2: Stacked autoencoders. When encoding data, AE-1 has to be computed first, while when decoding data, AE-2 comes first.

1.3 Convolving

AEs can be convolved, i.e., it is possible to compute the output of *a single* AE at different locations in a neighborhood (typically in a grid-like pattern, often with overlapping patches). The outputs of the AE are collected in a 3D array called encoded representation, as shown in Figure 1.3. In this document, we call convolution width and convolution height the number of times the AE is applied horizontally and vertically (2 and 2 in Figure 1.3).

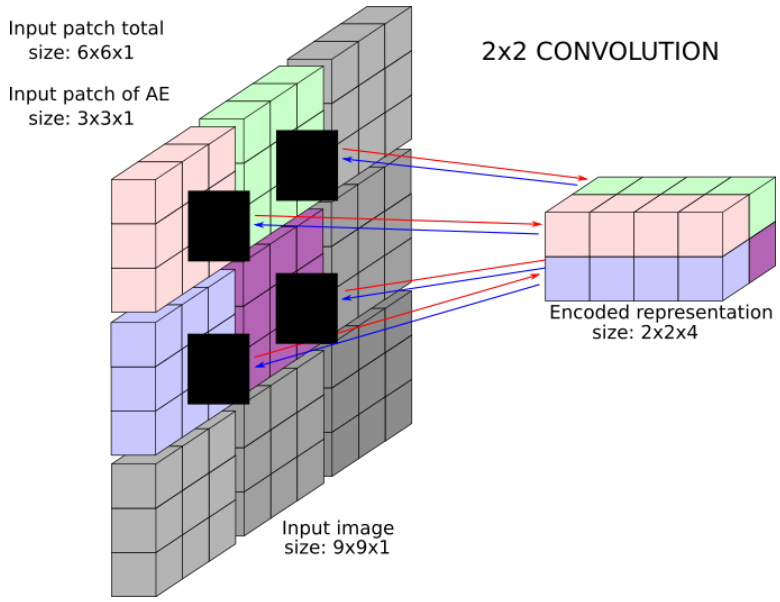

Figure 1.3: 2×2 Convolution of an AE with input 3×3, resulting in a $2 \times 2 \times 4$ array. Note that the gray parts in the image are disregarded in this example.

As seen in Figure 1.3, the area covered by the convolution does not necessarily have the same size has the input image. For example, in case of pixel-labelling tasks, it is usually sufficient if the pixels being classified as well as some neighborhood are covered by the convolution. However, when classifying the whole content of an image, then the whole image has to be used for input.

Take note however that using SCAEs on high-resolution images can be very time-consuming. The computational cost of a neural network is roughly linearly proportional to its number of inputs multiplied by its number of outputs. Convolutional Neural Networks (CNN) or SCAE are less expensive, as a single output of a layer is not connected to all inputs of that layer, but nevertheless the computational cost can be high when large areas are covered.

1.3.1 Convolution Offset

We call convolution offset the space between two AEs in a convolution. The offset is defined with two numbers H and V which specify how many pixels should be left horizontally and vertically between two input patches of the AE. In the literature, this is also often called "stride". The reference point

for the input patch of an AE is the top left corner, as ilustrated in Figure 1.4.

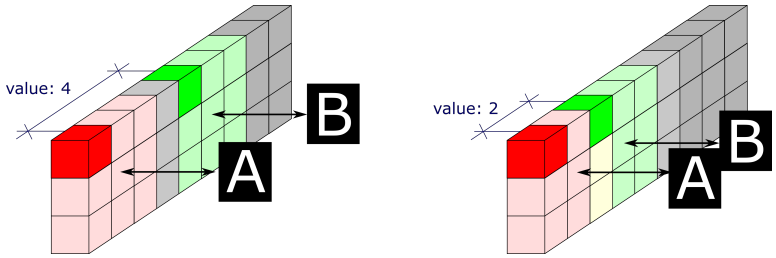

Figure 1.4: Examples of horizontal convolution using an AE with a 3×3 pixels input patch. The red and green correspond to two locations, A and B, of the AE in the convolution. The full color squares represent the reference point of the AE in location A or B. In the left case the offset is 4 pixels, leading to a distance of 1 pixel between two AE locations. In the right case the offset of 2 pixels leads to an overlap of 1 pixel on the 3rd column.

What is illustrated in Figure 1.4 applies also for the vertical axis. There are two special cases. First, if the offset value is equal to the input dimensions of the AEs, then there is no overlapping and not distance between the input patches (like in Figure 1.3). Second, if the offset value is 0 it means the AE will be applied always on the same place: this is a mistake which should lead the library to raise an exception.

1.4 Stacking & Convolving Autoencoders

The main idea of this framework is to convolve and stack AEs, as depicted in Figure 1.5, in order to get different levels of features which can then be used for machine learning purposes. The main questions for setting up an SCAE are:

- Which input patch size is optimal for each layer?
- How many features should there be in the different layers?
- Which convolution width & height should be used?
- Which convolution horizontal & vertical offsets should be chosen?

There seem to be no magical formula for answering accurately to these questions, thus to the best authors' knowledge, only experimentation can really do this. Take note that as the weights of the AEs are initialised

randomly, small accuracy differences might not be significant unless you run the tests several times. In such a case, we recommend to use the t-test from DIVASERVICES[3] [22] which allows to run significance tests in an extremely simple way.

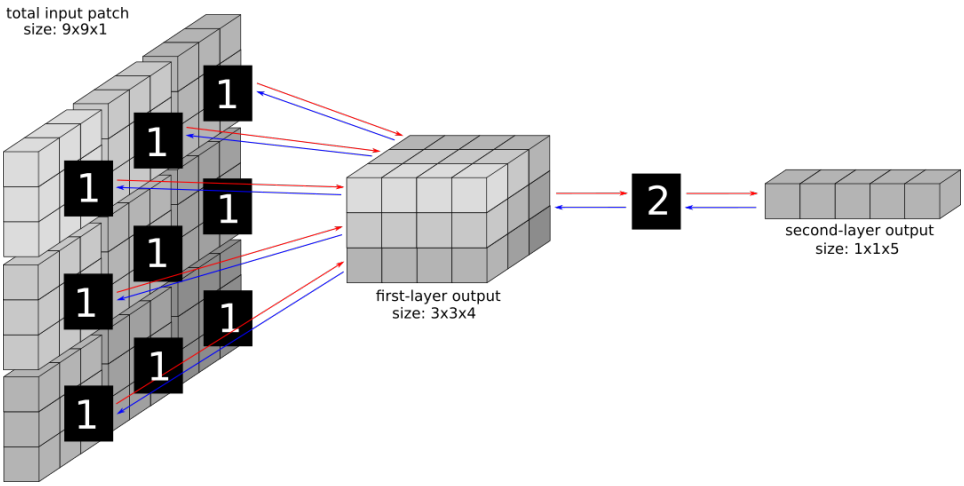

Figure 1.5: A stacked convolution autoencoder composed of two layers.

1.4.1 Determining Sizes

When building a stack of AEs, each layer corresponds to the convolution of an AE. The top layer however is a 1×1 convolution, i.e., its AE is computed at only one location, as it can be seen in Figures 1.5 and 1.6.

Assuming that we have a N-layers stack, then, for adding the $N+1$-th layer the size of the convolution in layer N has to be increased because it must have an output of the same dimension as the input of the $(N+1)$-th layer. The effect of resizing the size of the N-th layer is that the $(N-1)$-th must also be resized, and so on for all layers. For this reason, adding a layer to an SCAE might have a huge impact on its architecture. While users do not have to implement this backward size selection themselves, it is very important to understand it well when designing new architectures.

Figures 1.5 and 1.6 depict the convolution resizing. For simplicity reasons, offsets of the same size as the inputs are used in order to avoid overlapping. The first layer AE takes as input 3×3 pixels and outputs 4 values. The second convolution takes $3 \times 3 \times 4$ values as input, thus when adding a second layer the first layer has to be convolved 3×3 times as shown in Figure 1.5.

The third layer, as depicted in Figure 1.6 requires an input of $2 \times 2 \times 5$ values.

[3] http://divaservices.unifr.ch/ttest/

Table 1.1: Definition of SCAEs in Figures 1.5 and 1.6. While the input size corresponds to only one layer, the number of pixels covered by an AE depends also on the previous layers.

layer	1	2	3
input size	3×3	3×3	2×2
offset	3×3	3×3	–
covered pixels	3×3	9×9	18×18
hidden	4	5	3

This means that the second layer must be convolved 2×2 times. The second layer then requires $6 \times 6 \times 4$ inputs, which is obtained by convolving 6×6 the first layer.

Convolution sizes and offsets for a convolution layer are always based on its input. In Figure 1.5, the second layer has an input size of 3×3, which corresponds to having a 3×3 convolution in the first layer, and *not* to pixels in the image. Table 1.1 shows the difference between layer input size and pixels covered by the SCAE shown in Figures 1.5 and 1.6.

1.4.2 Training Stacked Autoencoders

Training a deep neural network can be computationally expensive. Although recent methods have been shown to lead to little or no vanishing gradient, computing new weights for a full network can be time-consuming. To tackle this issue, SCAEs are trained layer by layer during their construction. Before adding a new layer to an SCAE, its (current) top layer is trained to encode and decode its input with little error. Already-trained layers are not trained anymore.

1.5 Classification

The features learned by an SCAE correspond to the output of the top-layer AE. Three options are possible:

- They can be extracted and given to an external classifier (e.g., an Support Vector Machine (SVM)),
- Classification layers can be added on top of the SCAE, turning it into a classifier.
- The features at the center of each CAE of the SCAE can be concatenated and given to an external classifier.

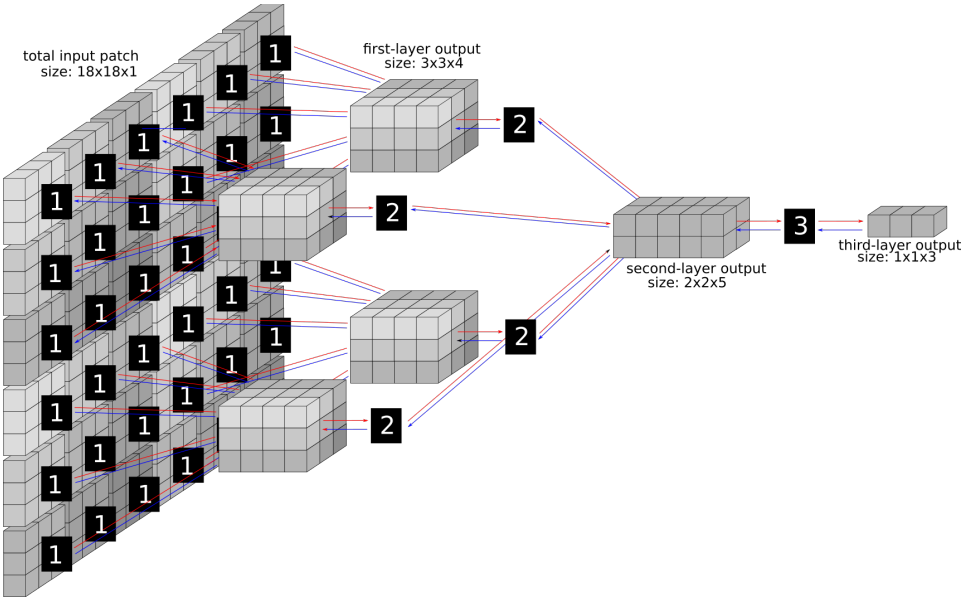

Figure 1.6: A stacked convolutional autoencoder composed of three layers. Note that it was obtained by adding a layer to the SCAE depicted in Figure 1.5.

The first option is closely related to methods based on hard-coded features extraction followed by classification. An example about how to do this is given in Section 2.2.

The second option is more interesting, as it allows to backpropagate classification errors not only through classification layers, but also through feature-extraction layers in order to fine-tune the features for a classification task.

The third option is a novel approach existing to our best knowledge only in *N-light-N*.

Chapter 2

Quick Start

This chapter contains the minimal information for using the framework as it is. More details about its component or its use are provided in the other chapters. We will first present how to train a CAE, then how to compute features from images using the CAE from a Java application.

2.1 Training a Convolutional Autoencoder (CAE)

The compiled `jar` file is a standard Java application, which does not require anything exotic to be installed on your computer, so you can just start it from the command-line:

```
1 java -Xmx8G -jar gae.jar my-training-script.xml
```

The `-Xmx8G` option tells the virtual machine that it can use up to 4 GB of memory. Depending on the task, you might want to increase or decrease this. When the memory is not or barely sufficient, then Java's garbage collector will use a lot of CPU power trying to free some memory – it is usually better to allow the use of more memory than necessary.

The `-jar` option with parameter `gae.jar`, tells the virtual machine which `jar` file should be executed.

Finally, the `my-script.xml` indicates which script should be executed. Some sample scripts are provided with *N-light-N* which you can use, and adapt if needed.

Please make sure that both the path of the jar and the one of the script are correct before launching the program. Note that you should avoid using absolute paths if possible.

2.2 Extracting Features

Once you have created and trained a CAE, extracting features from an image is a simple task which you can easily do from a Java application. You simply have to:

1. Load the CAE,
2. Load the image,
3. Let the CAE know where it should input data,
4. And ask it to compute a feature vector.

These 4 tasks correspond to the 4 following lines of Java code.

```
1 SCAE myScae = SCAE.load("scae.ae");
2 DataBlock inImg = DataBlock.dataBlockFromImage("input.png");
3 myScae.centerInput(inImg, centerX, centerY);
4 float[] features = scae.forward();
```

This feature vector can then be used for machine learning purposes. Take note that the array returned by the `forward()` method is allocated only once – if you want to store several feature vectors you will have to clone it.

2.3 The OSX Special Case

By default, Mac OSX seems to use the 1.6 version of Java, which dates from 2005[1]. The framework requires Java 1.8 (or more recent), therefore you might have to install the most recent version of Java and set your IDE/OS to use it. Some users have even indicated that installing Java 1.8 is not sufficient – a manual uninstallation of Java 1.6 could potentially be required.

[1] Apple wants you to develop only for OSX & IOS.

Chapter 3

Framework Architecture

This chapter provides an overview of the framework's architecture. Interactions among different entities are explained and a brief description of the main work flow is provided. If you intend to work *on* the framework, i.e., to extend or modify it (see chapter 6), you will need a very good knowledge about all of this.

3.1 Parsing Scripts

The class responsible parsing scripts and executing them `XMLScript.java`. When the framework is launched as stand-alone application, an `XMLScript` object is created and a map between command names and their implementation is generated. Then, the `XML` script is loaded and parsed. Invalid `XML`s (e.g., if closing tags are missing) will be rejected at this point. Then the content of the script is processed as shown in Figure 3.1 and described below:

(1) The children of the `XML`'s root element are processed in the same sequence as they appear in the script.

(2) Commands matching the child's names are retrieved from the map – if none is found then an error will be raised and the application will close.

(3) The command then receives the tag as parameter so that it has access to its attributes and children, and is then executed.

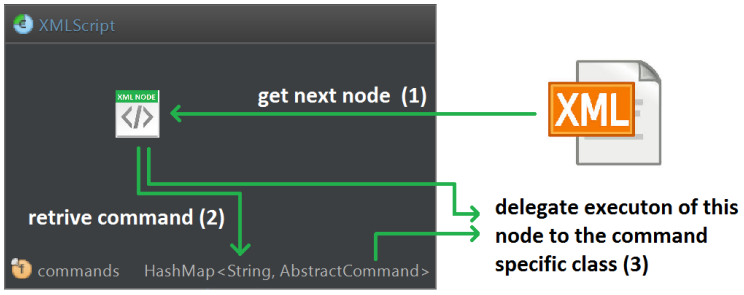

Figure 3.1: Command execution procedure.

3.1.1 XMLScript

This class is the core for parsing the XML. As mentioned in Section 3.1, it contains a map linking all command names to the corresponding command objects. These objects also get a reference to the `XMLScript` object in order to be able to access to the resources, e.g., datasets or SCAEs.

Each resource is is stored in a map and can be retrieved with its ID. The list of resources and information which the different commands might need is given below.

1. `Image.Colorspace colorspace`:
   stores which color space is being used by the script. Commands must take this into account where needed.
2. `Map<String, Dataset> datasets`:
   in lists of `DataBlocks` which can be used for training purposes.
3. `Map<String, NoisyDataset> noisyDataSets`:
   special datasets storing not one `DataBlock` per image but two – one clean and one noisy version.
4. `Map<String, SCAE> scae`:
   stores the SCAE currently in memory.
5. `Map<String, Classifier> classifiers`:
   stores the different classifiers currently in memory.

When creating new commands, it is mandatory to take the colorspace into account, even if not needed for the current experiment. One never knows what will happen next.

New commands must be added in the constructor of `XMLScript` following the same procedure as for the other ones.

3.1.2 AbstractCommand

All commands must inherit from `AbstractCommand`. This class allows not only to add new commands to the script, regardless of their nature, with minimum effort. It also provides some methods to extract and process data from `XML` elements. When implementing a new command, additionally to having to pass the reference to the `XMLScript` reference to the constructor (hint: think about the `super` keyword), two methods have to be implemented:

`execute(Element)`
responsible for carrying out the functionality the command provides. Here is where all the magic of the framework happens. The `XMLScript` object will call this passing the fetched node as parameter.

`tagName()`
returns the name of the command. This name must be a string which will perfectly match the name of the corresponding tag in the `XML` script. Names must respect general `XML` tag naming rules. Additionally, in order to keep the command names consistent, they should follow the form `word1-word2-...`, without capital letters. For instance command is is named `load-dataset`.

3.2 Data Structure

The whole framework works with the data structure of `DataBlock`. These blocks of of data are used as input for the neural networks, between layers and as output for neural networks. Users of the framework *will* have to deal with `DataBlock`s – fortunately they are very easy to use.

The following subsections present the `DataBlock` class, and its subclass `BiDataBlock`.

3.2.1 DataBlock

The DataBlock is an object which "simply" stores a 3D matrix of floats. Images can also be seen as 3D data, as they have a width, an height, and a given number of channels (e.g., 1 for grayscale images or 4 for CMYK images). The size of these dimensions is not fixed, i.e., the width, height and depth can have any value greater than 0 – as long as there is enough available memory.

The output of a convolution can be stored as an $A \times B \times C$ `DataBlock`, where A is its width, B its height and C its depth, or number of channels.

The output of a single AE can be stored as an $1 \times 1 \times C$ `DataBlock`. This is well illustrated in Figure 1.5.

The framework uses `DataBlocks` instead of standard 3D arrays of floats in order to provide additional functionalities, including loading and saving `DataBlocks`, extracting areas from them into 1D float arrays, and "pasting" data into them with overlapping.

In the following parts of this section, we will present how to read and write data in a `DataBlock`. In Creating & Loading `DataBlocks`, the `DataBlock` constructors are presented. In Data Access, methods which are simple but should be sufficient for almost all uses are presented. Unless you need to modify deeply the framework, knowing these I/O methods is probably sufficient. In Non-Image Data, how to load custom data is explained. This might be very useful if you do not intend to work on image data. In Extracting & Pasting Patches, some additional methods to transform `DataBlock` patches to and from float arrays. And finally, in Weighted Paste, we explain how to cope with overlapping data, i.e., how to take into account that overlapping in convolutions can lead to a single value to be constructed from two different sources.

Creating & Loading `DataBlocks`

`DataBlocks` has several constructors:

Listing 3.1: DataBlock constructors

```
1 public DataBlock(int width, int height, int depth)
2 public DataBlock(Image src)
3 public DataBlock(BufferedImage src)
4 public DataBlock(String fName) throws IOException
```

The first, and most generic constructor, allows to create a `DataBlock` with given dimensions. Should the user want to work on 1D or 2D data, some dimensions of the `DataBlock` can be set to 1. When storing individual samples in `DataBlocks`, we would recommend for performance issues to set the width to 1 when dealing with 2D data, and both the width and the height to 1 when dealing with 1D data.

Data Access

The dimensions of a `DataBlock` can be accessed with the following methods:

Listing 3.2: Access to DataBlock dimensions

```
1 public int getWidth()
```

```
2 public int getHeight()
3 public int getDepth()
```

It is possible to access to single values in a `DataBlock`. The coordinates of a value are first indicated with the depth (corresponding to a color channel when the `DataBlock` contains an image, and to a feature when it is the output of an AE), and then x and y coordinates:

Listing 3.3: Access to a single value

```
1 public float getValue(int channel, int x, int y)
2 public void setValue(int channel, int x, int y, float v)
```

It is also possible to directly access to all channels corresponding to a given set of coordinates using float arrays. The following two accessors do not work "by copy" – the array returned by `getValues` is the one actually stored in the `DataBlock`. Also the array passed as parameter of `setValues` will replace the one stored in the `DataBlock`.

Listing 3.4: Access to all channels at a given location

```
1 public float[] getValues(int x, int y)
2 public void setValues(int x, int y, float[] z)
```

Non-Image Data

`DataBlocks` have some constructors to load image data. The `XML` scripts allowing to train networks such data, and if you intend to use different kinds of data as input for neural networks, you will have to implement a subclass of `DataBlocks`. For this, it is highly recommended to read Chapter 5 and, depending on what you intend to do, also Chapter 6.

As indicated in Section 3.2.1, `DataBlocks` consist in a 3D array of floats, so any data that fits in such an array can be loaded into a `DataBlock`.

If you have 2D data (similar to a grayscale image), then it is more efficient to project this data along the height and depth axes of the `DataBlock`. Let us assume that you have a function $F(x, y)$ that returns the value of your data at coordinates (x, y), and that your data has a size of $w \times h$. This could be done, e.g., by reading from a file. Then, you can load this into a `DataBlock` in this way:

Listing 3.5: Loading 2D data into a DataBlock

```
1 DataBlock db = new DataBlock(1, h, w);
2 for (int x=0; x<w; x++) {
3   for (int y=0; y<h; y++) {
```

```
    db.setValue(x, y, 0, F(x,y));
  }
}
```

For 1D data, the values should be projected along the depth axe of the `DataBlock`. Assuming that you have a function $F(x)$ that returns the x-th value of your data, and that your data is composed of w values, then you can load it into a `DataBlock` in the following way:

Listing 3.6: Loading 1D data into a DataBlock

```
DataBlock db = new DataBlock(1, 1, w);
for (int x=0; x<w; x++) {
  db.setValue(x, 0, 0, F(x));
}
```

This second case is probably the most likely to happen, as useful feature vectors extracted from data are sometimes already available, or when data of inconsistent size has to be mapped to a fixed-size space. For example, currently experiments are being made on classifying edges of graphs that represent the structure of document pages [20]. For classification with *N-light-N*, we compute a fixed-length vectorial representation of an edge that captures the structure of its local neighborhood (subgraph). Hence, a standard AEs can be used.

Extracting & Pasting Patches

SCAEs take as input *patches* in `DataBlocks`. A patch correspond to the values of all channels in a rectangular area of a `DataBlock`. For simplifying read/write accesses to patches from layers composing SCAEs, the `DataBlock` class offers several useful methods.

Listing 3.7: Extracting a patch from a DataBlock

```
public void patchToArray(
  float[] arr,
  int posX,
  int posY,
  int w,
  int h
)
public float[] patchToArray(
  int posX,
  int posY,
  int w,
  int h
```

```
13 )
```

The first version of `patchToArray` puts values from a patch starting at coordinates (`posX`, `posY`) and of size `w`×`h` into the (already-existing) array `arr`. The array must have a size of `w`×`h`×`d`, where `d` is the depth of the `DataBlock`.

The second version of `patchToArray` works in a similar way, but allocates the returned array and returns the result. It might be slower, but does not require to have an already-existing array.

The reverse operation, i.e., copying an array to a patch, can be done with the following method:

Listing 3.8: Extracting a patch from a DataBlock

```
1 public void arrayToPatch(
2   float[] arr,
3   int posX,
4   int posY,
5   int w,
6   int h
7 )
```

where the values in the array `arr` are copied in a `w`×`h`×`d` patch, where `d` is the depth of the `DataBlock`.

Note that the layers composing an AE (see Section 3.3.1) work with float arrays extracted by the `patchToArray` method. This extraction is done by the AEs (see Section 3.3).

Weighted Paste

The convolutions in SCAEs might have some overlapping, which means an input value might be used more than one time. This also means that when decoding data, an SCAE might transmit several values to the same coordinates of a `DataBlock`. The solution to deal with this is to accumulate these values, count for each location of the `DataBlock` how many values were accumulated, and later divide the values by this count in order to get their mean. This count, which we call weight, is stored as floats which are shared for all channels of the `DataBlock`.

Using weighted paste and normalizing data, i.e., dividing values by the corresponding weights, can be done with the following methods:

Listing 3.9: Pasting with weight

```
1 public void weightedPaste(DataBlock source, int x, int y)
```

```
public void weightedPaste(
  float[] arr,
  int from,
  int x,
  int y
)
public void weightedPatchPaste(
  float[] arr,
  int posX,
  int posY,
  int width,
  int height
)
public void normalizeWeights()
public void clear()
```

The first `weightedPaste` pastes a `DataBlock` into the target; this source `DataBlock` can be at most as big as the target. The second method pastes at a given location an array which length must be equal to the depth of the `DataBlock`. The third method pastes an array onto a patch – the length of this array must be equal to the number of values in the patch.

The `normalizeWeights` method divides all values by the corresponding weights, and sets all weights to 1. Finally, the `clear` method can be used to set all values and weights of a `DataBlock` to 0. This last method should always be called before starting to paste data with weights.

BiDataBlock

The `BiDataBlock.java` is an extension of `DataBlock.java`. The difference is that instead of a 3D matrix of floats, the data is stored in a `BufferedImage` (hence the name: **B**uffered**I**mage `DataBlock`). The advantage of using this instead of a normal `DataBlock` is that it occupies less space in memory and is loaded faster. The drawback is that is it slower when being used, as `RGB` codes have to be turned into floats. It is possible to write float values in a `BiDatablock` with the different accessors, however these floats will be approximated as only 256 different values can be stored.

3.2.2 Dataset

Datasets are collections of `DataBlock`s. They are used by `XML` scripts, but can also be used outside of it. They implement the generic interface `Collection<DataBlock>`. Data is shuffled every time the `iterator` me-

thod is called, so iterating several times on a `Dataset` will not be done in the same sequence.

NoisyDataset

It is possible for a `Dataset` to contain two versions of each sample, one clean and one noisy. They can be used for training Denoising Autoencoders (DAEs).

3.3 Autoencoder Building Blocks

Autoencoders (AEs) are the core of the framework. They are the building block for SCAEs, which are then used for building classifiers.

In this framework, AEs act as an interface between the high level entities (such as SCAEs, classifiers) and the low level layers. The AE stores references to four `DataBlock`s : *input*, *output*, *previous error* and *error*, which are detailed below.

INPUT this is a reference to the input `DataBlock`. For the first layer this usually corresponds to an input image (which has typically a depth of 1 or 3, depending on whether the image is in grayscale or in color). For the other layers, the input corresponds to the output `DataBlock` of the previous layer and has depth equal to the number of features generated by the previous layer. The AE takes an input patch which might be smaller than its input `DataBlock`, and stores the coordinates of the top left corner of this input patch, as well as its width, height and depth. These coordinates (x,y of the top left corner) are the same for the *previous error* datablock, but are different for *output* and *error*.

OUTPUT this is a reference to the output `DataBlock`. If the AE is not convoluted this `DataBlock` has size $1 \times 1 \times n$, where n is the number of outputs of the AE's encoder. Otherwise, the next layer of the SCAE determines the size of this `DataBlock`, as the input patch size of the next layer specifies how many times the current layer has to be convoluted. See Section 1 for more details. The depth of the output `DataBlock` corresponds to the number of outputs of the encoding layer of the AE; for each position in the convolution of the SCAE corresponds a position in the output `DataBlock`. Thus, the encoding-layer requires only a reference to the corresponding 1D float array within the output `DataBlock`.

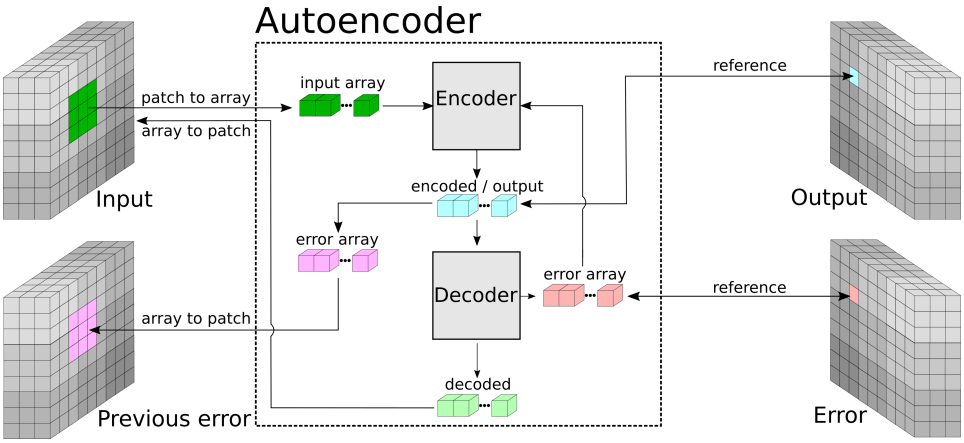

Figure 3.2: Data flow in an autoencoder. Note that *input*,*output*, *previous error* and *error* are `Datablocks`. The size (especially the depth) of the `DataBlocks` on the left of the figure might differ from the size of the ones on the right.

ERROR this is a reference to a `DataBlock` storing the error of the AE, relatively its encoding layer. When trained for encoding and decoding, or when trained for classification purposes, errors are backpropagated to the encoding-layer of the AE. These errors are stored in this `DataBlock`. The error `DataBlock` has the same size as the output one; when trained, the AEs read their error in this `DataBlock` at the same coordinates as they write in the output `DataBlock`.

PREVIOUS ERROR this is a reference to the error `DataBlock` of the previous layer of the SCAE or classifier. For the first layer, this reference is null, thus accessing to this `DataBlock` should be done carefully. This `DataBlock` is used when backpropagating classification errors.

As we can see in figure 3.2 an AE is composed of two main parts: an encoder and a decoder. These two objects implements `Layer` the interface, which is detailed in Section 3.3.1, or extend the `AbstractLayer` class which already provides the implementation of some methods defined in the interface. The encoder and decoder of an AE do not necessarily have to be of the same class. This opens the possibility to create different kinds of AEs by simply providing them different kind of layers.

The abstract class `AutoEncoder` provides the basic functionalities for a normal behaving AE (meant as the typical definition found on books and de-

scribed in Chapter 1). The `AutoEncoder` itself is used for managing its two layers by connecting them together correctly, managing inputs and outputs, and delegating calls of `encode` and `decode` methods, so the encoding itself is not implemented in the `AutoEncoder` class.

The concrete class `StandardAutoEncoder` implements `AutoEncoder` and provides constructors allowing to select the `Layer`s by specifying their class name. The list of existing layers is given in Section 3.4.1.

For creating different kinds of AE which do not follow the standard encode-decode methodology, please refer to section 6.3 of chapter 6.

3.3.1 Layers

The two layers contained by an `AutoEncoder` must implement the `Layer` interface. As shown in Figure 3.3, all inputs and outputs are arrays of floats and not `Datablock`. Subclasses of `AutoEncoder` have to provide these arrays correctly to the different `Layer`s, which means that when implementing a new kind of `Layer` it is not necessary to care about complicated data structures.

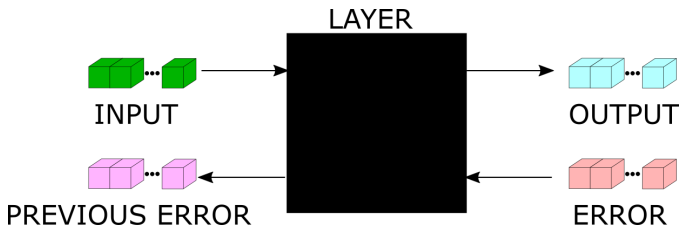

Figure 3.3: Interface of a layer. Notice how all inputs and outputs are array of floats and not `Datablock`, which means creating new layers does not require dealing with complex data.

The following methods below are present in the `Layer` interface. While there seem to be a *frightening* lot of them[1], most are only accessors which are extremely easy and fast to implement and in most cases the abstract class `AbstractLayer` can be used (see Section 3.3.1).

In any case, the overhead of implementing this interface compared to the implementation of the encoding or decoding method can be considered as negligible. Listing 3.10 gives the list of the most important methods.

Listing 3.10: Some important methods from the Layer interface

```
1 void setInputArray(float[] inputArray);
2 void setOutputArray(float[] inputArray);
```

[1]There are 23 of them, which can be considered as a lot.

```
3  void setError(float[] error);
4  float[] getError();
5  void setPreviousError(float[] prevError);
6  void compute();
7  void setExpected(int pos, float expectedValue);
8  float backPropagate();
9  float learn();
10 Layer clone();
```

These methods are described below:

`setInputArray` Indicates to the `Layer` from which array data will be read. Note that it might be called once or several times, so any class implementing the `Layer` interface should keep a copy of this reference.

`setOutputArray` Indicates to the `Layer` to which array it should write data. Note that it might be called once or several times, so any class implementing the `Layer` interface should keep a copy of this reference.

`setError` Indicates to the `Layer` where it will have to read its errors during the training phase. Note that it might be called once or several times, so any class implementing the `Layer` interface should keep a copy of this reference.

`getError` Returns a reference to the error array.

`setPreviousError` Indicates to the `layer` where it should backpropagate errors to. This does *not* correspond to the output error of a previous layer, but to an array which is generated by the `AutoEncoder` class (see Figure 3.2).

`compute` This is the main, and potentially most interesting, method of the class. When it is invoked, the `Layer` should compute the outputs with regard to the inputs.

`setExpected` This method is used during the training phase. It indicates for a given output the values which would have been expected. An error can be computed for each output by subtracting the expected value to the actual output. This error should be stored in the error array. Take note that depending on how the training is done, this method might not be called – it might be possible that errors will be written directly to the error array.

`backPropagate` This method should have to effects. First, the error stored in the error array should be backpropagated to the previous `Layer`. Backpropagation is usually used only in artificial neural networks and is mathematically well defined. Depending on what the `Layer` is, this backpropagation might be implemented in a different way – try to estimate how by how much the inputs should be modified in order to decrease slightly the output error, and transmit it to the previous error array. Second, the error array should be used to compute how to modify the parameters of the `Layer`. In the case of a neural network, it corresponds to computing the error gradients. In order to allow batch training, these values should be accumulated and not applied immediately. The method should return the mean output error.

`learn` This method is invoked when the parameters of the `Layer` have to be updated. It might be done after each `backPropagate` call, or after several ones.

AbstractLayer

the abstract class `AbstractLayer` already implements some of the methods of the `Layer` interface. This class can be used as "starting point" for most kinds of standard layers.

The only abstract methods in this class are `compute`, `backPropagate`, `learn` and `clone`.

3.4 Package Overview

As there are many classes in the framework, it might be worth know where they are, and what kind of classes are in the different packages. The first tier of packages is composed by *ae* (autoencoder) and *diuf.diva.dia.ms*. In the former there is only the `Main` class[2], while in the latter there is the framework, divided in three sub categories: `ml` (machine learning), `script` and `util`. They are briefly presented below.

[2]This class should be left untouched, and the only arguments which should be provided are the names of the `XML` scripts to execute.

3.4.1 `diuf.diva.dia.ms.ml`

This is the core of the framework. The classes contained in this package and subpackages are related to machine learning.

The single interface present in this package is `Classifier`, which all types of classifiers of the framework should implement if possible in order to allow easy comparisons between them.

`diuf.diva.dia.ms.ml.ae`

The `ae` package is supposed to contain `AutoEncoder`s. Should an user implement a new kind of AE, the corresponding class should be placed in this package.

There here is also a subpackage for every type of higher level construct (SCAE, classifier, ...):

aec is the package which contains all classes relative to the AEClassifier, a classifier using features from all layers of an SCAE.

ffcnn is the package which contains all classes relative to the Feed Forward Convolutional Neural Network. These classes are the `ConvolutionLayer` (which clones AEs and "convolve" them) and the core `FFCNN` classes. Note that while called "neural network", the `FFCNN` can be composed of anything extending the `Layer` interface correctly.

ccnn is the package which contains all classes relative to the Combined Convolutional Neural Network. This work in progress is not accessible from scripts, and does not implement the `Classifier`. The reason for this weak integration within the framework is that it allows to combine several SCAEs taking each possibly a different `DataBlock` as input. Including this into scripts could lead to spaghetti code, which nobody would ever want to work with.

mlnn is the package which contains all classes relative to Multi-Layered Neural Network (MLNN). This is used in the AEClassifier.

scae is the package containing all classes relative to SCAEs: the `Convolution` (which convolve the AEs) and the core `SCAE` classes.

diuf.diva.dia.ms.ml.layer

This package contains the classes which do most of the neural network's work: the layers. Layers can be used as encoder, decoder, or even general-purpose "neural" network layer. This package contains two base elements,

the `Layer` interface, and the `AbstractLayer` abstract class. They are described with more details in Section 3.3.1.

Additionally, it contains the implementations of the `Layer` interface. It is mandatory to place new ones in this package, as the full class name (including packages[3]) is used when creating new SCAEs from `XML` scripts.

The list below contains the list of currently existing layers.

NeuralLayer this is a typical Neural Network (NN) layer. It uses a soft sign activation function and can be trained with back-propagation.

LinearLayer this is a linear associator layer which does not have an activation function. The output is given simply by $\vec{b} + \vec{w} \times \vec{x}$, where $\vec{b}$ is the bias vector, $\vec{w}$ the set of weights and $\vec{x}$ the input. This layer can be trained with back-propagation as well. At the moment, as the weights are never re-scaled it might happen that some of them grow uncontrollably and lead to float overflows. Fixes for this behavior are currently being investigated.

OjasLayer this is a linear associator layer which does not have an activation function. The output is given simply by $\vec{b} + \vec{w} \times \vec{x}$, where $\vec{b}$ is the bias vector, $\vec{w}$ the set of weights and $\vec{x}$ the input. This layer can be trained with the Oja's rule algorithm, which is a modified version of Hebb's Rule. This algorithm makes the weights of the layer converge towards a value that computes the principal components of the input. There is a parameter γ to find by trial & error such that the weights do not diverge in a similar fashion as they do with the naive implementation of a linear associator.

ReLU this is a rectified linear unit, i.e., it uses the activation function $f(x) = 0$ for $x < 0$, and $f(x) = x$ otherwise. This is a special case of the more generic similar activation function introduced by Mhaskar *et al.* [23]. An activation cost of $10^{-3} \cdot f(x)^2$ is used, as well as a weight decay of 10^{-3}. These parameters can currently be modified only from `ReLU.java`.

SoftPlusLayer this is a continuous version of the rectified linear unit using the following activation function: $f(x) = \ln(1 + e^x)$. As it is bounded in $]0, \infty[$, it cannot be used as decoder for images, which are encoded in $[-1, 1]$.

[3] Nevermind, just make sure they're all in there.

SigmoidLayer this is the well known sigmoid activation function defined as $f(x) = (1 + e^{-x})^{-1}$. As it is bounded in $]0, 1[$, it cannot be used as decoder for images, which are encoded in $[-1, 1]$.

3.4.2 diuf.diva.dia.ms.script

This package contains the `XMLScript` class.

Its subpackage *diuf.diva.dia.ms.script.command* contains the different commands, which all inherit from `AbstactCommand`. New commands should of course be added at the right place.

3.4.3 diuf.diva.dia.ms.util

This package contains classes which serves as support for the framework:

DataBlock
This is the class used for storing data in the framework. It is described in Section 3.2.1.

BiDataBlock
This is a subclass of `DataBlock` storing image data in `BufferedImage` It is described in Section 3.2.1.

DataBlockDisplay
This is an abstract class allowing to display on the screen `DataBlock`s having either three channels (interpreted as `RGB` data) or one channel (interpreted as grayscale data).

FeatureDisplay
Implementation of the `DataBlockDisplay` used to display the features learned by an SCAE. Every time the `update` method is called, features will be extracted and displayed.

Dataset
This is a set of `DataBlock`s which can be used for training SCAEs. It is described in Section 3.2.2.

NoisyDataset
This is a set of noisy and clean `DataBlock`s which can be used for training DAE. It is described in Section 3.2.2.

Image
This class is used for managing colorspaces. It is used for loading images, and provides methods (e.g., `toCMYK`) for switching from a colorspace to another one. `DataBlock`s can be created out of `Image`s. The class

`diuf.diva.dia.ms.util.Image` should not be mixed up with the `Image` class from `java.awt`.

PCA
This class[4] is used for performing Principal Component Analysiss (PCAs) on data.

Tracer
This class allows to draw error plots when training machine learning methods.

[4]Which is based on https://github.com/mkobos/pca_transform

Chapter 4

Scripts

This chapter explains how to write a custom XML script for the framework. Proper usage of all implemented commands is described.

4.1 Why XML Scripts?

The nature of the framework is lightweight and modular. Hence the user should not write Java code for running an experiment, but rather use scripts. Thereafter are mentioned the main advantages of using the script:

- *No framework knowdledge*: using the framework requires no knowledge about it. In fact, it is only required to write a very simple script to benefit of the most complex functionality implemented.
- *No coding knowledge*: it is possible to use the framework with no coding skills. The fact that the framework is implemented in Java is completely transparent to the user.
- *Easy storage*: experimental code is often erased for newer versions, while scripts can be easily stored everywhere.
- *Easy variation*: after a simple copy-paste of the script is possible to make a variations of it while not modifying a single bit in the framework.
- *Repeatability*: since scripts they are easily stored, it is also easy to repeat experiments later. We recommend strongly users not to update scripts, but to make new versions of them in order to be easily able to reproduce experimental results if needed.
- *Parallel runs*: it is extremely easy to have multiple experiments running at the same time.

4.2 Writing Scripts

A script is composed of a root tag (which name is unimportant) and a *sequence* of tag corresponding to instructions. Each instruction has its own tag. Instructions parameters are composed of attributes or children tags. An *Hello World* script is as follows:

Listing 4.1: Base script - colorspace i always mandatory

```
1 <say-hello colorspace="RGB">
2   <print>Hello !</print>
3 </say-hello>
```

Take note commands are executed one at a time and sequentially as they are written into the script. For this reason, initialization of data is usually done at beginning of the script. For example, you first need to load the dataset and an SCAE before training it or adding a layer to it. The following sections describe the different commands which have been so far implemented.

4.3 Loading & Saving

This section explains how to load and save entities such as datasets, `SCAE`, classifiers or any kind of saved files which is supported from the framework.

4.3.1 Loading a Dataset

Datasets are sets of images. The user can then use these images as they like. For example, one can load a dataset and use it as:

- training dataset
- evaluation dataset
- ground truth dataset in case of pixel-labelling tasks

The role of datasets is not specified and its determined from the context of execution. For instance, when training an SCAE the framework will just assume that the provided dataset is a training dataset and use it as such. In case the dataset provided was not a training dataset (for example a ground truth one) no errors will arise and the duty of find out this semantic mistake is left to who wrote the script.

Mandatory fields

Datasets are referred to using an ID which must be specified as attribute of the parent tag.

<**folder**> specifies the path to the images. It can be both relative or absolute path. Note that if relative path can be given relatively to the current working directory.

<**size-limit**> specifies the maximum number of images which are loaded. This can be useful when using a computer with a low quantity of available memory, when using large datasets, or to test quickly a script. If the size limit is set to 0, then all images are loaded.

Listing 4.2: Loading a dataset - mandatory fields

```
1 <load-dataset id="stringID">
2   <folder>stringPATH</folder>
3   <size-limit>int</size-limit>
4 </load-dataset>
```

Optional Fields

Optionally it is possible to specify that the dataset has to be composed of buffered data blocks (see Section 3.2.1) by adding the <buffered/ > tag. The advantage of the `BiDataBlocks` is that they save memory as they are buffered, and the disadvantage is that data access is much slower.

<**buffered/**> If present buffered data blocks will be used instead of normal data blocks.

Moreover, it is possible to generate several versions of the `BiDataBlocks` at different scales.

<**scales**> parent tag for one or more <scale>. Take note that when loading several scales of an image, the size limit of the dataset will be loaded for all scales. That is, if you have a limit of n images, n different images are loaded for each scale, having a total number of $n \cdot \#scales$ images loaded.

<**scale**> specifies what is the desired scaling factor. For example with 0.5 as value, the dataset will be loaded and downscaled to half of his original size.

Listing 4.3: Loading a dataset - full parameters

```
<load-dataset id="stringID">
  <folder>stringPATH</folder>
  <size-limit>int</size-limit>
  <buffered/>
  <scales>
    <scale>float</scale>
    <scale>float</scale>
    [..]
    <scale>float</scale>
  </scales>
</load-dataset>
```

Loading a Noisy Dataset

Datasets with two versions of each image, one with degradations and the other without, can also be loaded in order to train Denoising Autoencoders (DAEs). Noisy datasets cannot yet be composed of `BiDataBlocks`, nor store several scales of the data. Two folders have to be specified:

Listing 4.4: Loading a noisy dataset - full parameters

```
<load-dataset id="stringID">
  <clean-folder>stringPATH</clean-folder>
  <noisy-folder>stringPATH</noisy-folder>
  <size-limit>int</size-limit>
</load-dataset>
```

4.3.2 Unloading a Dataset

In certain situation, the possibility of unloading a dataset which is no longer used will come handy in order to free some memory. For example, after training a classifier it is possible to unload the training sets and load the evaluation sets. It is only necessary to specify the ID of the desired dataset in the attribute of the parent tag.

Listing 4.5: Unloading a dataset - full parameters

```
<unload-dataset id="stringID"/>
```

4.3.3 Loading an Entity

Datasets are not the only "things" which can be loaded from file. At the moment the framework can load SCAEs and classifiers as well. It is only necessary to specify the ID of the desired entity in the attribute of the parent tag.

<**file**> specifies the path to the entity. It can be both relative or absolute path.

Listing 4.6: Loading an entity

```
<load id="stringID">
  <file>stringPATH</file>
</load>
```

4.3.4 Saving an Entity

`SCAE` and classifiers can be saved to file and stored for further use. To use stored items in other scripts it is necessary to load them (see Section 4.3.3).

<**file**> specifies the path to the entity. It can be both relative or absolute path.

Listing 4.7: Saving something

```
<save ref="stringID">
  <file>stringPATH/output.dat</file>
</save>
```

4.4 Stacked Convolution AutoEncoders (SCAE)

This section explains how to create, add layers to, train and evaluate SCAEs.

4.4.1 Creating a SCAE

Stacked Convolutional Autoencoders (SCAEs) are initially created with a single layer and convolution is only executed when more layers are added later. Every layer should be trained before adding a new one on top of it. For more details about that, or if the reader is not familiar with convolution principles, reading Chapter 1 is advised.

Mandatory Fields

SCAEs are referred to using an ID which must be specified as attribute of the parent tag.

<**unit**> what type of unit should be used for encoding/decoding. Different units might have different syntax, but in every case the parent tag would define the units type (in the example below "standard") and the children the specific parameters for that unit (in the example how many hidden nodes we want and what kind of layers should be used to encode/decode).

<**width**> width of the input patch in pixels

<**height**> height of the input patch in pixels

<**offset-(x,y)**> by how much the AE has to be shifted when this layer is convolved. If these values match the width and height of the input patch there will be no overlapping when convolved. Note these values are completely meaningless if this will be the only layer (no further layers means no convolution!). Values smaller than one are not allowed.

Listing 4.8: Creating a simple SCAE with standard autoencoder

```
<create-scae id="stringID">
 <unit>
   <standard>
     <layer>stringClassName</layer>
     <hidden>int</hidden>
   </standard>
 </unit>
 <width>int</width>
 <height>int</height>
 <offset-x>int</offset-x>
 <offset-y>int</offset-y>
</create-scae>
```

Syntax For Different Units

The syntax for the different units is described in the following paragraphs.

standard: Standard autoencoder. It has real inputs and outputs.

<**layer**> specifies what type of layer should be used to encoder and decode. Note that in this field the *exact* name of the Java class without extension should be used. For example "NeuralLayer" is a correct value and "NeuralLayer.java" is not. In section 3.4.1 the possible values for this field are presented.

<**hidden**> defines how many hidden neurons have to be used.

Listing 4.9: Syntax of standard unit

```
<unit>
  <standard>
    <layer>string</layer>
    <hidden>int</hidden>
  </standard>
</unit>
```

DAENN: De-noising autoencoding Neural Network. It has real inputs and outputs.

<**hidden**> defines how many hidden neurons have to be used.

Listing 4.10: Syntax of DAENN unit

```
<unit>
  <DAENN>
    <hidden>int</hidden>
  </DAENN>
</unit>
```

BasicBBRBM: Binary-binary RBM, basic implementation. As it requires binary inputs, the previous layer should have a binary output.

<**hidden**> defines how many hidden units have to be used.

Listing 4.11: Syntax of BasicBBRBM unit

```
<unit>
  <BasicBBRBM>
    <hidden>int</hidden>
  </BasicBBRBM>
</unit>
```

BasicGBRBM: Gaussian-binary RBM, basic implementation. It has real inputs and binary outputs.

<**hidden**> defines how many hidden units have to be used.

Listing 4.12: Syntax of BasicGBRBM unit

```
<unit>
  <BasicGBRBM>
    <hidden>int</hidden>
  </BasicGBRBM>
</unit>
```

PCA: Principal Component Analysis. It has real inputs and real outputs. Instead of using random initial weights, a PCA algorithm is used for initializing them. It is possible to drop dimension or to keep them all as well.

<**layer**> specifies what type of layer should be used to encoder and decode. Note that in this field the *exact* name of the Java class without extension should be used. For example "NeuralLayer" is a correct value and "NeuralLayer.java" is not. In section 3.4.1 are presented the possible values for this field.

<**dimensions**> how many hidden neurons (consider this as the dimensionality of subspace projected by the PCA)

Listing 4.13: Syntax of PCA unit

```
<unit>
  <PCA>
    <layer>stringClassName</layer>
    <dimensions>full|int</dimensions>
  </PCA>
</unit>
```

4.4.2 Training a SCAE

Training an SCAE requires a dataset, but there is no need for a ground truth as AEs are unsupervised machine learning methods.

Mandatory Fields

As SCAEs are referred to using an ID, here is necessary to provide a reference ID as attribute to the <train-scae> method. The execution stop as soon as only one of the two conditions (maximum number of samples, maximum running time) is met.

<**dataset**> The ID of the dataset which has to be used for training.

<**samples**> maximum number of samples to use during the training. Samples are image patches selected randomly. A value of 0 means "no limit".

<**max-time**> maximum training time in minutes. A value of 0 means "no limit".

Listing 4.14: Training an SCAE - mandatory fields

```
<train-scae ref="stringID">
  <dataset>my-dataset</dataset>
  <samples>int</samples>
  <max-time>int</max-time>
</train-scae>
```

Optional Fields

While training SCAEs is possible to log some information. It is possible to watch the features of the AE and the recording (code/decode) of an image

result as they change in time during training. It can be very useful for getting an idea of what is being learned, and how well features describe data. There is also the possibility to show a plot reporting the training error of the SCAE epoch wise. It is also possible to save this plot to file.

<**display-features**> if present, features will be showed during training. The numeric value specifies the frequency of update for the frame showing the features (every how many samples shall the features be updated). A good value is $\approx \frac{1}{1000}$ the number of samples which will be used during the training.

<**display-recoding**> if present, recoding will be showed during training. The string value specifies the path of the image which will be displayed. It is *highly* recommended to use a very small image, otherwise processing it might take much more time than the training itself.

<**display-progress**> if present, the plot with training error will be shown during training.

<**save-progress**> if present, the plot with training error with be save at the location specified.

Listing 4.15: Training an SCAE - full parameters

```
<train-scae ref="stringID">
  <dataset>stringID</dataset>
  <samples>int</samples>
  <max-time>int</max-time>
  <display-features>int</display-features>
  <display-recoding>stringPATH</display-recoding>
  <display-progress>int</display-progress>
  <save-progress>stringPATH</save-progress>
</train-scae>
```

4.4.3 Adding a Layer to an Autoencoder

Adding a layer to an AE uses almost the same syntax as creating one. Beware that when a layer is added the whole previous structure gets convoluted as many times as the input of the new layer. Refer to the Chapter 1 for more information.

Mandatory Fields

As SCAEs are referred to using an ID, it is necessary to provide the reference (ID) of the SCAE that we want to add a layer to as attribute <add-layer>.

<**unit**> the type of unit should be used for encoding/decoding. See section 4.4.1 for more information about the different units.

<**width**> width of the input patch, not in pixels, but sampled on the outputs of the previous layer.

<**height**> height of the input patch, not in pixels, but sampled on the outputs of the previous layer.

<**offset-(x,y)**> by how much the AE has to be shifted when this layer will be convoluted. If these values match the width and height of the input patch there will be no overlapping when convoluted. Note these values are not used if this will be the only layer (no further layers means no convolution!). Values smaller than one are not allowed.

Listing 4.16: Adding a layer to an SCAE

```
<add-layer ref="stringID">
  <unit>
    <Standard>
      <type>NeuralLayer</type>
      <hidden>int</hidden>
    </Standard>
  </unit>
  <width>int</width>
  <height>int</height>
  <offset-x>int</offset-x>
  <offset-y>int</offset-y>
</add-layer>
```

4.4.4 Recode Images & Store The Result

It is possible to visualize the reconstructed image after coding/decoding with an SCAE. This could be useful to have a quick idea on the performance of the SCAE for this task.

Mandatory Fields

As SCAEs are referred to using an ID, it is necessary to provide the reference (ID) of the `SCAE` we want to evaluate as attribute <recode>.

<**dataset**> reference ID of the dataset that will be reconstructed (coded/decoded).

<**destination**> specifies the path where the reconstructed images will be saved. It can be both relative or absolute path.

Listing 4.17: Showing the recoded image of an SCAE - mandatory fields

```
<recode ref="stringID">
  <dataset>stringID</dataset>
  <destination>stringPATH</destination>
</recode>
```

Optional Fields

Optionally is possible to specify a desired offset for the operation. By default the offset would be the size of the input patch. Ideally with a offset of 1 all pixels would be reconstructed and we would obtain the most precise recoding possible. However, the more layers the `SCAE` has, the slower the computation will be. This becomes a concern with big images, where the computation time can easily become prohibitive. In case the recoding take too much time it is suggested to leave the default value. Note that with the default value if the `SCAE` has more than one layer the final image will present "patterns" which are the consequence of the offsetting with a value higher than 1.

<**offest-(x,y)**> this specifies the x and y offset of the recoding.

Listing 4.18: Showing the recoded image of an SCAE - full parameters

```
<recode ref="stringID">
  <dataset>stringID</dataset>
  <offset-x>int</offset-x>
  <offset-y>int</offset-y>
  <destination>stringPATH</destination>
</recode>
```

4.4.5 Show Learned Features

It is possible to visualize the feature learned by an SCAE after the training. For visualizing the features during the training please refer to Section 4.4.2.

Mandatory Fields

As SCAEs are referred to using IDs, it is necessary to provide the reference (ID) of the SCAE which features we want to visualize as attribute to <show-features>.

<**scale**> scaling factor. It stretches the images without interpolation and makes them more comfortable to look at without zooming. Note that large values might lead to huge images being generated.

<**file**> specifies the path where the features image will be saved. It can be both relative or absolute path.

Listing 4.19: Showing features after training

```
<show-features ref="stringID">
  <scale>int</scale>
  <file>stringPATH</file>
</show-features>
```

4.5 Classifiers

4.5.1 Creating a Classifier

The framework allows the creation of classifier starting from an already existing (and trained!) `SCAe`. As of now, two different kinds of classifiers are supported by the framework: Auto Encoder Classifier (AEC) and Feed Forward Convolutional Neural Network (FFCNN).

Mandatory Fields

Classifiers are referred to using an ID which must be specified as attribute of <create-classifier>.

<**type**> specifies which kind of classifier should be built. A description of the different classifiers is provided later in this section. The possible values for this field are `AEClassifier` and `FFCNN`. It *must* be written literally so (case sensitive and typos' free).

<**layer**> specifies what type of layer should be used to encoder and decode in the classification layers. Note that in this field the *exact* name of the Java class without extension should be used. For example "NeuralLayer" is a correct value and "NeuralLayer.java" is not. In section 3.4.1 the possible values for this field are presented. Currently, only the `FFCNN` can use this option, so it is not mandatory to `AEClassifier`s.

<**scae**> reference ID of the `SCAE` which will be used to build (initialize) the classifier.

<**neurons**> indicates the number of neurons in the different hidden layers, separated by commas. For example, a value of "4,7" means that there will be two hidden layer with respectively 4 and 7 neurons. This value is "unbounded", meaning that there is no restriction on the amount of the hidden layers one can add. Beware that the more layers the slower the classifier will be (among other issues!). If this tag is present, then at least one number must be provided.

<**classes**> indicates how many classes the classifier will work with. This number also corresponds to the number of neurons in the output layer.

Listing 4.20: Creation of a classifier

```
<create-classifier id="stringID">
  <type>enum{AEClassifier,FFCNN}</type>
  <layer>stringClassName</layer>
  <scae>stringID</scae>
  <neurons>int,[int]*</neurons>
  <classes>int</classes>
</create-classifier>
```

Autoencoder Classifier: The `AEClassifier` is a combination of an SCAE and a neural network which takes as input the Central Multilayer Feature (CMF) of the SCAE. During the training, the SCAE is not fine-tuned. This leads to a much faster training, and the CMF allows to classify small-scale details in the center of the input patch.

Feed Forward Convolutional Neural Network: this classifier is built out of an SCAE. Several instances of classes implementing the `Layer` interface are added to perform the classification task. While training the

classification layers, the AEs also gets trained per back propagation. In the convolutions, the AEs are cloned instead of applying the same at different locations. Thus, they can learn different weights at different locations during the training as classifier and get therefore more adapted to the task[1].

4.5.2 Training a Classifier for Pixel Labelling

One of the most recent features of the framework is the possibility of training a classifier for pixel labelling scripts. All classifiers which can be created with a script can also be trained for pixel labelling in the same script if desired. Note that it currently works only when working with `RGB` images.

Mandatory Fields

As classifiers are referred to using an ID, it is necessary to provide the reference (ID) of the classifier which we want to train as attribute <train-classifier>. The execution stop as soon as only one of the two conditions (maximum number of samples, maximum running time) is met.

<**dataset**> The ID of the dataset containing input images which have to be used for training.

<**groundTruth**> The ID of the dataset which has to be used as ground truth for training. The `RGB` code of each pixel must match its class number, i.e., `0x000000` for the class 0, `0x000001` for the class 1, ...

<**samples**> Maximum number of samples to use during the training, i.e., how many training steps[2] A value of 0 means "no limit".

<**max-time**> Maximum training time in minutes. A value of 0 means "no limit".

Listing 4.21: Training a classifier - mandatory fields

```
1 <train-classifier ref="stringID">
2   <dataset>stringID</dataset>
3   <groundTruth>stringID</groundTruth>
4   <samples>int</samples>
5   <max-time>int</max-time>
6 </train-classifier>
```

[1] In "FFCNN" the term "convolution" is slightly misused as the training "deconvolves" the layers.

[2] Gradient descents in case of standard neural networks.

Optional Fields

While training a classifier is possible to log some information in a plot reporting the training error of the classifier epoch-wise. It is also possible to save this plot to a file.

<**display-progress**> If present, the plot with training error will be shown during training.

<**save-progress**> If present, the plot with training error with be saved at the specified location.

Listing 4.22: Training a classifier - full parameters

```
<train-classifier ref="stringID">
  <dataset>stringID</dataset>
  <groundTruth>stringID</groundTruth>
  <samples>int</samples>
  <max-time>int</max-time>
  <display-progress>int</display-progress>
  <save-progress>stringPATH</save-progress>
</train-classifier>
```

4.5.3 Evaluating a Classifier

After having trained a classifier for pixel labelling, it is possible to evaluate how well it performs directly from a script. All classifiers which can be created and trained with a script, can also be evaluated in the same script if desired. With single-class evaluation the accuracy metric will be measures, whereas with the multi-class evaluation precision and recall will be computed per class.

Mandatory Fields

As classifiers are referred to using an ID, here is necessary to provide the reference (ID) of the classifier which we want to evaluate as attribute of <evaluate-classifier>.

<**dataset**> The ID of the dataset which has to be used for training.

<**groundTruth**> The ID of the dataset which has to be used as ground truth for training. It must have the same format as when training a classifier (see Section 4.5.2).

<**offest-(x,y)**> In order to accelerate tests, it can be possible to skip some of the pixels. With an offset of 1, all pixels will be classified and evaluated, with an offset of 2 only 25% of them will be, ... In order to get an idea about how good a classifier is performing, it is not necessary to evaluate all the pixels. It is possible to compare classifiers when the same test data and the same offsets are used. In case of large images, selecting offsets leading to $\approx$ 10'000 classifications for each image, should lead to already rather accurate evaluations.

<**method**> specifies whether the evaluation should be single or multi class. The possible values are `single-class` and `multiple-classes`.

`Single-class`: Only the output class with the highest response get considered, then it is either the correct one or not. The output is an image (green/red only) for each picture in the dataset.

`Multiple-classes`: all the outputs of the classifier gets evaluated and the error is computed for all of them. The output is an image for each class for each picture in the dataset.

<**output-folder**> specifies the path where the images representing the evaluation will be stored. It can be both relative or absolute path. On these images, the colors have to be interpreted as follows:

- `BLACK`: True negative
- `BLUE`: False negative
- `RED`: False positive
- `GREEN`: True positive

Listing 4.23: Training a classifier - mandatory fields

```
<evaluate-classifier ref="stringID">
  <dataset>stringID</dataset>
  <groundTruth>stringID</groundTruth>
  <offset-x>int</offset-x>
  <offset-y>int</offset-y>
  <method>enum(single-class,multiple-classes)</method>
  <output-folder>stringPATH</output-folder>
</evaluate-classifier>
```

4.6 Utility

4.6.1 Beep

If the operating system allows it, it is possible to make a beep on demand. This might be useful to warn the user that a long procedure (like training an SCAE or a classifier) is over. It uses default system sound.

Listing 4.24: Beep !

```
1 <beep/>
```

4.6.2 Printing Text

It is possible to output text to the standard output. It comes handy to display the current action being performed, the training error, the evaluation reconstruction or anything you might want to know.

<**print**> the content that you want to print

Listing 4.25: Printing text

```
1 <print>string</print>
```

4.6.3 Describing an Entity

Sometimes, it might be useful to describe the content of an SCAE, a dataset or a classifier. This command prints to screen useful information about the selected entity. Provide the ID of the entity which you want to be described.

Listing 4.26: Describing something

```
1 <describe ref="stringID"/>
```

4.6.4 Variables

As the scripts are meant to describe simple procedures, it does not seem necessary to make a complex script language with loops, expressions, conditions and such. New procedures should rather be included into the framework; for example a new training methodology could be implemented as a new type of AE or as a new instruction. The script however contains something similar to variables, which can be used for different purposes[3].

[3]For example when saving an AE, including its training accuracy in the file name.

Mandatory Fields

As entities are referred to using an ID, here is necessary to provide a reference (the ID) of the variable as attribute of the parent tag.

<**define**> the content of the variable

Listing 4.27: Defining a variable

```
1 <define id="stringID">string</define>
```

Usage

Variables can be defined and their name is replaced in all values and options (but not tag names!) by a search-and-replace method when the script is executed.

Listing 4.28: Setting and using a variable

```
1 <test-script>
2   <define id="MYVARIABLE">World</define>
3   [..]
4   <print>Hello MYVARIABLE!<print/>
5 <test-script>
```

It is not possible to replace tag names with variables. For example, the next example is **incorrect**:

Listing 4.29: **Incorrect** use of variables

```
1 <test-script>
2   <define id="display">print</define>
3   [..]
4   <display>Hello World !</display>
5 <test-script>
```

4.6.5 Storing a Result

It is possible to store the result of an instruction in a variable. All instructions return a `String`. For most of them it is empty, but other return useful information; for example, when you train an SCAE, the training error is returned. The result of the last instruction is displayed if the scripts exits without error. Each instruction also stores its result in the `$ANS` variable, so you can use it *immediately* after the instruction. If you need to use it later, or twice, you have to store it.

Mandatory Fields

As entities are referred to using an ID, here is necessary to provide a reference (the ID) of the variable where the result will be stored as attribute of the parent tag.

Listing 4.30: Storing a result

```
1 <store-result id="MY_VAR"/>
```

Usage

Listing 4.31: Example of storing a result

```
1 <test-script>
2     <display>Hello World !</display>
3     <store-result id="MY_VAR"/>
4     <print>The previous command returned MY_VAR</print>
5 </test-script>
```

4.6.6 Colorspace

The framework can be used with different colorspaces. So far, no evaluation has been done in order to see if a colorspace is more interesting than another one for a given task. Only one colorspace can be used in a script; it is not possible to change it during the execution of the script. The colorspace is defined with the `colorspace` attribute of the root element of the script, as shown in Listing 4.32. If no colorspace is defined, then the RGB colorspace is used.

Listing 4.32: Script using the YUV colorspace

```
1 <test-script colorspace="YUV">
2     <print>Hello World !</print>
3 </test-script>
```

The different colorspaces currently available are:

RGB red, green and blue components of the pixels. It is the default colorspace.

XYZ it corresponds to a linear transformation of the RGB colorspace.

YUV luminosity, U chrominance and V chrominance. It is a linear transformation of the RGB colorspace.

HSV hue, saturation, value.

CSSV it is a transformation of the HSV colorspace. The hue, which is cyclical, is replaced by both its sine and cosine. This colorspace has therefore four dimensions.

CSS this is a modification of the CSSV colorspace, where the two first parameters are multiplied by the value, and the value is removed. It is therefore a three-dimensions colorspace.

CMYK corresponds to the cyan, magenta, yellow and black components of the color. It is the colorspace which is usually used by printers.

GRAYSCALE this is a one-dimension colorspace obtained by averaging the RGB values. If you use images which are are already in grayscale, you still **must** specify to use the grayscale colorspace, otherwise they will be loaded as RGB images.

All images are transformed to the correct colorspace automatically when loaded. Take note that it also defines the depth of the input of the first layer of SCAEs; when used in Jana programs, the same colorspace transformation has to be made. For this, you can use the instances of `Image` as parameter of the `DataBlock` constructor (see Sections 3.2.1 and 3.4.3).

Chapter 5

Using the Framework as a Java Library

N-light-N can be used both as a stand-alone executable to run `XML` scripts, or as a Java library. In the latter case, the full potential of *N-light-N* is revealed, as all classes become available and can be combined to run experiments and to deal with data that is not handled by the scripts.

The following sections present how to use the most important classes of the Framework.

5.1 `DataBlocks`

`DataBlocks` have been presented in Section 3.2.1. Here, some usage examples are given.

5.1.1 Loading from Images

Listing 5.1: Loading a DataBlock from an image, 3 cases

```
1 // Case 1: using the file name
2 DataBlock db = new DataBlock("input.jpg");
3 // Case 2: using a buffered image
4 BufferedImage bi = ...;
5 DataBlock db = new DataBlock(bi);
6 // Case 3: using an Image and changing the colorspace
7 Image img = new Image("input.jpg");
8 img.toCMYK();
9 DataBlock db = new DataBlock(img);
```

5.1.2 Creation from Scratch

When dealing with different kinds of data format, it is possible to fill manually a `DataBlock`. The following example creates a $10 \times 20 \times 30$ `DataBlock` and fills it with random values in $[0, 1[$.

Listing 5.2: Creating and filling a DataBlock

```
int width = 10;
int height = 20;
int depth = 30;
DataBlock db = new DataBlock(width, height, depth);
for (int x=0; x<width; x++) {
  for (int y=0; y<heiht; y++) {
    for (int z=0; z<depth; z++) {
      db.setValue(z, x, y, (float)Math.random());
    }
  }
}
```

Should you use a specific data format, you can subclass `DataBlock` and implement constructors for loading this format.

5.1.3 Saving & Loading

As most class in the framework, `DataBlock` uses the `Serializable` interface, which means `Object` streams can be used for loading and saving `DataBlock` instances.

Listing 5.3: Saving and loading DataBlocks

```
// Saving 'myDataBlock'
ObjectOutputStream oos = new ObjectOutputStream(
    new FileOutputStream("out")
);
oos.writeObject(myDataBlock);
oos.close();
// Loading a datablock
ObjectInputStream ois = new ObjectInputStream(
    new FileInputStream("input")
);
myDataBlock = (DataBlock) ois.readObject();
ois.close();
```

Additionally, the `DataBlock` class offers two static methods for loading `DataBlocks`, and two non-static ones for saving them:

Listing 5.4: I/O methods for DataBlocks

```
public static DataBlock load(String fname)
public static DataBlock load(ObjectInputStream ois)
public void save(String fname)
public void save(ObjectOutputStream oos)
```

5.2 SCAE

SCAEs created and trained with `XML` scripts can be loaded, but it is also possible to create them manually. In this section, we first present how to load SCAEs, how to construct them manually, how to train them, and how to extract features with them.

5.2.1 Loading & Saving SCAEs

As it is the case for most classes of the framework, the `SCAE` class implements `Serializable` and can thus be easily saved and loaded with object streams. Two methods are present for this:

Listing 5.5: I/O methods for SCAEs

```
public void save(String fileName)
public static SCAE load(String fileName)
```

Here is an usage example:

Listing 5.6: Loading and saving an SCAE

```
mySCAE.save("output.ae");
SCAE otherSCAE = SCAE.load("output.ae");
```

5.2.2 Creating an SCAE

As explained in Chapter 1, SCAEs are created and trained layer-by-layer. Creating an SCAE and adding a layer to it have almost the same syntax and are therefore presented together.

The constructor and `addLayer` method take three parameters as input:

1. An `AutoEncoder`,
2. The horizontal offset when convolving it,
3. The vertical offset when convolving it.

An example is given below:

Listing 5.7: Creating an SCAE

```
SCAE scae = new SCAE(
  new StandardAutoEncoder(5, 5, 3, 10, "NeuralLayer"),
  1,
  2
);
```

In this example, the first layer correspond to an AE using standard neural layers, taking as input a $5 \times 5 \times 3$ patch, outputting 10 features. When convolved, the AE will offset by 1 pixel horizontally, and 2 pixels vertically.

Adding a layer follows the same syntax. Take note that the AE does not know about the SCAE, so you have to specify as input depth the number of outputs of the previous layer – in this case 10:

Listing 5.8: Adding a layer to an SCAE

```
scae.addLayer(
  new StandardAutoEncoder(3, 3, 10, 20, "NeuralLayer"),
  1,
  2
);
```

5.2.3 Training SCAEs

The layers of the SCAEs are trained one after another, i.e., you need to train a layer before adding a next one. Assuming that some `DataBlock`s are available, the training is done by repeating the following operation many times:

1. Select a random `DataBlock`
2. In this `DataBlock`, select a random location such that the SCAE can take it as input without border problems,
3. Set the input of the SCAE to that location,
4. And finally, call the `train` method.

This is illustrated below:

Listing 5.9: Training an SCAE

```
Random rand = new Random();
```

```
for (int sample=0; sample<limit; sample++) {
  // Select DataBlock
  DataBlock db = getRandomDataBlock();

  // Select location
  int x = rand.nextInt(
    db.getWidth() - scae.getInputPatchWidth()
  );
  int y = rand.nextInt(
    db.getHeight() - scae.getInputPatchHeight()
  );

  // Set input
  scae.setInput(db, x, y);

  // Train
  scae.train();
}
```

5.2.4 Extracting Features

To extract features corresponding to the input patch at a given location, there are two options:

1. The features learned by the top layer are returned by the `compute` method,
2. The Central Multilayer Feature (CMF) can be extracted by the `getCentralMultilayerFeatures` method.

This is illustrated below, assuming that `db` is an input `DataBlock`, and that we want to extract features from a patch centered on (x,y):

Listing 5.10: Extracting features

```
scae.centerInput(db, x, y);
float[] topFeatures = scae.forward();
float[] cmf = scae.getCentralMultilayerFeatures();
```

Note that if you want only the CMF array, then you can ignore the return value of `forward`, but calling this method is still needed. Also the arrays returned by both methods are *always* the same; each `forward` call will overwrite them. If you need to store several feature arrays, then you should clone them, as illustrated below:

Listing 5.11: Extracting and storing features

```
List<float[]> lst = new ArrayList<>();
for (int x=0; x<10; x++) {
  for (int y=0; y<10; y++) {
    scae.centerInput(db, x, y);
    lst.add(
      scae.forward().clone()
    );
  }
}
```

5.3 Classifiers

N-light-N contains several classifiers, which work in different ways although they are all based on SCAEs. They are described in this section.

5.3.1 FFCNN

The `FFCNN` is a feed-forward convolutional neural network which is based on an SCAE and can be trained for classification tasks. This is done by:

1. Adding one or several fully-connected layers on top of the SCAE,
2. And training the resulting network with backpropagation through all layers, including the ones coming from the SCAE.

Additionally, it is possible to "deconvolve" the `FFCNN`, i.e., allowing different weights to be learned for each position of the AEs in the convolutions. This increase much the number of parameters to learn, however it might increase the performance for some classification tasks, e.g., pixel labelling.

Creating an `FFCNN`

The `FFCNN` class offers several constructors, which are given below:

Listing 5.12: Constructors of the FFCNN class

```
public FFCNN(SCAE scae)
public FFCNN(SCAE base, String layerName, int nbClasses)
public FFCNN(
  final SCAE base,
  String layerClassName,
```

```
6     final int nbClasses,
7     int[] additionalLayers
8 )
```

The first one will create an FFCNN without adding new layers to the SCAE. It might be useful to plug the FFCNN into a CCNN (see Section 5.3.3).

The second, and probably most useful constructor allows to add a single classification layer on top of the SCAE. The type of layer has to be specified in the exact same way as for SCAEs; the list of available layers is given in Section 3.4.1. The number of classes (or outputs) has to be specified.

Finally, the third constructor adds the possibility to stack several layers, each with a different number of neurons. This number of neurons has to be specified in the last parameters. For example, to create an FFCNN with three additional layers, one with 80 neurons, one with 25 and an output layer with 2 neurons, the syntax would be the following:

Listing 5.13: Construction of an FFCNN

```
1 FFCNN ffcnn = new FFCNN(
2   myScae,
3   "NeuralLayer",
4   2,
5   new int[]{80, 25}
6 );
7 // Optional:
8 ffcnn.deconvolve();
```

5.3.2 AEClassifier

The AEClassifier is an experimental approach making use of Central Multilayer Feature (CMF) coupled to a multilayer neural network. Proper investigation has not been published yet, however preliminary results tend to indicate that this approach could lead to an accuracy similar to fine-tuned networks (see Section 5.3.1).

Creating an AEClassifier

This class has a single constructor:

Listing 5.14: Constructor of the AEClassifier class

```
1 public AEClassifier(
2   SCAE ae,
3   int nbClasses,
```

```
4   int... additionalLayers
5 )
```

The number of classes also corresponds to the size of the last layer of the `AEClassifier`; as there is no backpropagation done through the SCAE, it is recommended to add more layers to the classifier.

For example, an `AEClassifier` with three classification layers of 80, 25, and 2 neurons is constructed as following:

Listing 5.15: Construction of an AEClassifier

```
1 AEClassifier aec = new AEClassifier(
2   myScae,
3   2,
4   80,
5   25
6 );
```

The `AEClassifier` takes as input the Central Multilayer Feature (CMF) generated by the SCAE. These features have been applied successfully for pixel-level layout analysis [13], although we have shown that not all of them might be relevant for a classification task [17]. Although not properly shown yet, we can expect the CMF to perform better for the classification of small-scale elements than using the output of the top layer of an SCAE without fine-tuning.

5.3.3 CCNN

`CCNN` stands for "Combined Convolutional Neural Networks". It consists of a two-stages construct:

- A set of `FFCNN`s and/or `CCNN`s,
- Which outputs are given to a set of neural layers.

The different `FFCNN`s composing the `CCNN` can have different input sources, of different dimensionality. This is currently an experimental classifier which has not been thoughtfully investigated.

The constructor of the `CCNN` is the following:

Listing 5.16: Constructor of the CCNN class

```
1 public CCNN(CCNN[] ccnn, FFCNN[] ffcnn, int... topNeurons)
```

5.4 Training a Classifier

Training a classifier requires data and time. The required data is a set of inputs with the corresponding expected classification results. The networks are trained by presenting them with values and letting them know what was expected of them, and after a while, the networks should learn to give better results than random choices.

The classifiers, excepted the `CCNN`, implement the same interface which allows an easy training.

Let us assume that we have an array of `DataBlock`s (e.g., loaded from the images of the MNIST[1] dataset), and an array of integer corresponding to class labels (e.g., the digits corresponding to the MNIST images). Then the training can be done as follow:

Listing 5.17: Training a Classifier

```
// Load the data
DataBlock[] data = loadData();

// Load the labels
int[] label = loadLabels();

// Number of classes
int nbClasses = 10;

// Number of training epochs - can be different
int nbEpochs = 25;

// Load the classifier
Classifier classifier = loadClassifier();

// For a given number of epochs...
for (int epoch=0; epoch<nbEpochs; epoch++) {

    // We initialize a counter
    int nbErr = 0;

    // We go through the dataset
    for (int i=0; i<data.length; i++) {
        // We set the input of the classifier
        classifier.setInput(data[i], 0, 0);

        // We ask the classifier to compute
```

[1] http://yann.lecun.com/exdb/mnist/

```
28          classifier.compute();
29
30          // We ask for the class number (single class)
31          int res = classifier.getOutputClass(false);
32
33          // We count the errors
34          if (res!=label[i]) {
35              nbErr++;
36          }
37
38          // For all output, we set its expected value, which
39          // is 1 for the output corresponding to the expected
40          // class, 0 for the other ones
41          for (int j=0; j<nbClasses; j++) {
42              if (label[i]==j) {
43                  classifier.setExpected(i, 1);
44              } else {
45                  classifier.setExpected(i, 0);
46              }
47          }
48
49          // We backpropagate the error
50          classifier.backPropagate();
51
52          // And make the classifier learn
53          classifier.learn();
54      }
55
56      // We can display the number of errors
57      System.out.println("Epoch "+epoch+", nbErr="+nbErr);
58  }
```

Note that for simplicity reason, data loading code is not given. If you want more details about this, see Section 5.1. In this code, the classifier can be an instance of `FFCNN` or `AEClassifier`.

In case of a `CCNN`, the syntax would be similar, but with the following differences:

- The `CCNN` is supposed to use several input sources, not only a single `DataBlock` as in this example,
- And the `setInput` method gets an additional parameter, the `FFCNN` number in the `CCNN`.

5.5 Classification

Using a classifier is extremely simple once you know how to train it. It can be done in a similar fashion as in Listing 5.17, without the training phase (i.e., without loading labels, and stopping at line 31).

5.5.1 Advices

Training a classifier is both a simple and complicated task. The approach below is a bit naive and lacks some tricks which would probably increase the accuracy – some of them are given below.

Balance the Data

Some machine learning methods are lazy. If one of the classes appears extremely rarely, then the methods might learn never to output it. We recommend that if you have N classes, that you select $\approx \frac{1}{N}$ of the training data from each class.

Don't Wait for Hours

If the accuracy of a network does not seem to continue increasing after a while, it is usually unlikely that this will suddenly change. If the network does not reach a desired accuracy, then you should maybe change its topology, input size, ...

Don't Test Once

When networks are initialized with random weights and trained on randomly selected data, then the final accuracy will have some randomness. If you test two different approaches and notice that one is slightly better after a single test, it does not mean it will always be like this. Do tests several times to increase your level of confidence.

You can use the following tool, provided by the DivaServices [22], to compute levels of confidence from a set of results:

`http://divaservices.unifr.ch/ttest/`

This tool currently considers that higher results are better, so provide an accuracy measurement, and not an error rate.

Shuffle !

Do not train sequentially. Let us assume that you have a two-classes problem, and that you train first on N samples from the first class, and then on N samples from the second class. Then, we can make two observations:

1. You have balanced training data, which is good,
2. But as the last N training steps will ask only for the second class, the network might learn to output only it.

Shuffle your data, is possible between each epoch. For any short training sequence, there should be as much entropy in both the inputs and the expected outputs.

Chapter 6

Framework Upgrade

This chapter is a guide on how to implement new components for the framework. Do please read chapter 3 for getting an idea of the framework architecture before trying to implement something.

6.1 Creating your Own XML Command

This section explain how to implement a brand new commands which will be then possible to run with an XML script.

Where to Start

Every command *must* extend the `AbstractCommand` class. In the constructor, you have to pass the `XMLScript` reference to the super constructor, as shown in Listing 6.1. This is necessary to allow access access to all script resources. Now we only need to implement the two abstract methods `execute(Element element)` and `tagName()`.

Listing 6.1: Starting a new commands from scratch

```
1  public class myCommand extends AbstractCommand {
2    public myCommand(XMLScript script) {
3      super(script);
4    }
5
6    public String execute(Element element) {
7      // Implement your command functionality here
8    }
9
10   public String tagName() {
11     // Tag will be <newcommand></newcommand>
12     return "newcommand";
13   }
14 }
```

Implementing *tagName()*

This method must return the name of the command in the `XML`. It must not have the same name as an already-existing command. Additionally, it should follow the naming convention "no-capital-letter", with different words separated by a "-". Avoid special characters, spaces or digits.

Implementing *execute(Element element)*

This is the core of the command, so it is where we will implement the functionality that we want to carry out. The `XMLScript` will call this methods passing as parameter the node with the name provided in *tagName()*. From this node we can extract the children or attributes of our command. The abstract class `AbstractCommand` provides a couple of methods which are very useful to handle the `XML` nodes, so that you don't need to get your hands "dirty" with `JDom` or any other `XML` libraries. These methods are:

- `readAttribute(Element e, String name)`
  Reads an attribute of a tag
- `readElement(Element parent)` Returns the text contained into the tag. For example, if applied to <a>hello, it would returns "hello".
- `readElement(Element parent, String child)` Returns the content of the specified child of the parent element passed.

With these methods we can easily extract all the data we need to proceed. Note that if you do not use these methods, then you will have to take care of the variables yourself (see Section 4.6.4).

Example

The example below illustrates better how commands can be added. The new command makes a simple math operation, with an additional optional parameter to print something to screen.

The desired command is the following:

Listing 6.2: XML examlpe of the new command

```
<test>
  <new-command operator="plus">
    <first>1</first>
    <second>2</second>
    <text>Hello world</text>
  </new-command>
</test>
```

And the Java implementation is:

Listing 6.3: Example implementation of our new command

```
package diuf.diva.dia.ms.script.command;

public class MyCommand extends AbstractCommand {
  public myCommand(XMLScript script) {
    super(script);
  }

  public String execute(Element element) {
    // Fetching the attribute
    String operator = readAttribute(element, "operator");

    // Fetching operands
    int a = Integer.parseInt(readElement(element, "first"));
    int b = Integer.parseInt(readElement(element, "second"));

    // Compute operation
    int result;
    switch (operator) {
      case "plus":
        result = a + b;
        break;
      default:
        result = -1;
        System.out.println("Unknown operator");
    }

    // Testing if additional parameter is present
    String text=""; // empty by default
    if (element.getChild("text") != null) {
      text = readElement(element, "text");
    }

    // Print result
    script.print(
      "Result=" +
      result + " " + text);
    }
    return String.valueof(result);
  }

  public String tagName() {
    // Tag will be <new-command></new-command>
    return "new-command";
  }
}
```

Adding A Command To XMLScript

Even though we now have our perfectly working new command, the instance of `XMLScript` does not know about its existence. We then need to add our newly implemented command to the hard-coded list of available ones. To do so, just add the following line (listing 6.4) in the method *prepareCommands()* in `XMLScript.java`.

Listing 6.4: Adding out example to available command list

```
1 addCommand(new MyCommand(this));
```

6.2 Creating your Own Layer

In order to create your own layer, you need to properly implement the Java `Layer` interface. The abstract class `AbstractLayer`, which already implements most methods of the `Layer` interface, can also be used. This is a simple class which serves as "starting point" when creating a new kind of layer. Unless when implementing very special layers, it is recommended to use `AbstractLayer` rather than `Layer`. This way, one can focus on logic rather than utility methods as *setInput()* or so.

As a reminder, as shown in figure 6.1, the input, output, previous error and error are array of floats and not `Datablock`. This means that their value is passed by reference and not by value, hence you should store the references of these arrays and not make copies of them! It is now briefly presented what is exposed in the layer interface and what is their expected behaviour.

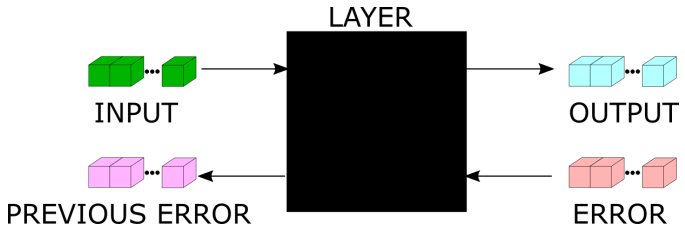

Figure 6.1: Interface of a layer. Notice how all inputs and outputs are array of floats and not `Datablock`, thanks to the work done by the `AutoEncoder` class.

Computing

As it names indicates, `compute` method is has to compute the output values using the inputs and the parameters learned by the layer. Writing to the output float array is sufficient, there is no need to care about the `DataBlock`s used by the SCAE.

Learning Related

The following three methods are the core of the layer training methodology. They implement the learning functionality and therefore, with *compute()* they define the behavior of the layer.

> `void setExpected(int pos, float expectedValue)` This method uses the expected value passed as parameter to compute the error of a given output. This is not necessarily the only way to set the error. This method is used in two cases: 1) the expected values of a decoder correspond to the input of the encoder, and 2) if the layer is on the top of a classifier, then the expected outputs correspond to the classification results.
>
> `float backPropagate()` This method corresponds, in a "standard" neural network[1], to an error backpropagation and gradient computation. This means that the error of the output has to be transmitted to the input (`previous error` in Figure 6.1). Additionally, modifications of the layer's parameters have to be planned. This means, in the case of a "standard" neural network, that gradients must not be subtracted to the weights at this point, but accumulated. This method might be called several times (batch training) before the `learn` method gets called.
>
> `float learn()` This method is used to apply parameter modifications accumulated by one or many `backpropagate` calls. Once it is done, values which were accumulated must be reset to 0.

Input Related

The following are methods which make it easy to handle the input of the layer.

> `float[] getInputArray()`
> This returns a reference to the input array.

[1] Some layers might work in a completely different way, but the terminology should be kept for clarity (and interface) reasons.

`void setInputArray(float[] inputArray)`
This set the array passed per parameters as new input array.

Output Related

The following are methods which make it easy to handle the output of the layer.

`int getOutputSize()`
This returns the size of the output vector. This must correspond to *getOutputArray().length*.

`float[] getOutputArray()`
This returns a reference of the output array.

`void setOutputArray(float[] inputArray)`
This set the array passed per parameters as new output array.

Error Related

The following getters & setters for the two `float[]` array representing *previous error* and *error*. Take care that everything should be handled by reference: getters should return a reference of the array and not a copy and setters should not copy the array before setting it!

`float[] getPreviousError()`

`void setPreviousError(float[] prevError)`

`float[] getError()`

`void setError(float[] error)`

Utiliy

`Layer clone()` It is of utmost importance that the layer can be correctly cloned, otherwise AEs cannot properly clone themselves.

6.3 Creating your Own Autoencoder

In order to create your own AE, you need to properly extend the Java abstract class `AutoEncoder`. To do so, at least the following methods have to be implemented or rewritten:

- `constructor()`
  Although not so special, the constructor is an important part. Here after calling
  `super(inputWidth, inputHeight, inputDepth, outputDepth)`,
  you need to provide the AE with the two `Layer` objects for encoder and decoder.

- `AutoEncoder clone()`
  This is the typical cloning methods. It is mandatory that it creates a deep copy of the AE because otherwise other entities (such as FFCNN) will have unexpected behaviors when using AE.

- `char getTypeChar()`
  This methods return a char indicating the type of the AE. For example the classic NN returns 'n'. This is useful for logging and printing purposes, especially when describing SCAEs.

When these methods are implemented, the new AE is ready to be initialized and used inside classifiers or SCAEs. Note that although you can create a new kind AE, in the sense that it did not exist before in the framework, by providing the two `Layer` objects (encoder, decoder layers) in the constructor, it could also be done with the `StandardAutoEncoder` class.

The main advantage of subclassing `AutoEncoder` is that any behavior can differ much if other methods are overwritten. For example, one could imagine some pre-processing of the input patch before putting it into the encoding layer's input array.

Keep in mind however, that an autoencoder is a delicate and well oiled machine. This means that you should override methods (like `learn()` or `train()`) only if you know what you are doing. Do not hesitate to add assertions at many places in order to make sure that everything goes well.

6.4 Creating your Own Classifier

In order to create your own classifier, it is recommended to properly implement the Java interface `Classifier`. Dong so would allow to simple comparison of classifiers, and to integrate it in the scripts for pixel-labelling tasks. The methods in the `Classifier` interface are detailed below.

Setting Inputs

One very important point for a classifier is setting the input correctly. There are in two methods for setting the input:

`void setInput(DataBlock db, int x, int y)`
In this methods `x` and `y` are the coordinate of the top left corner of the patch.

`void centerInput(DataBlock db, int cx, int cy)`
In this method, `cx` and `cy` are exactly the coordinates of the central pixel of the input patch..

The input width and input height hhave to be taken into account when using the classifier in order to avoid accessing to values which are outside of the input `DataBlock`. For example, on the x-axis it is clear that with `setInput()` you can start at 0, while with `centerInput()` you have to start at half the patch size at least.

Furthermore, in case of pixel labelling, be sure that the location you set as input corresponds to what you want. It sounds strange but consider that with `setInput()` the pixel under evaluation is NOT the one at the coordinates you provided to the method but one half patch down-right (as it is the center of the patch that gets evaluated). This can cause problems when comparing classification results to a ground truth if not handled carefully.

Computing

For the aspect of computing the result the interface is very simple. The `compute()` method has to be implemented. It has no return value, as other methods will be used for interpreting classification results.

`void compute()`
This method will execute the classifier and make him produce an output that can be subsequently fetched.

Getting the Output/Results

After computing the outputs, it is necessary to interpret them. The interface provides the following to methods to handles this.

`int getOutputClass(boolean multiClass)`
This method returns the classification result of the last evaluated input. The classification might be single or multi-class. In case of single class the result will be a normal integer indicating the class number. In case of multiclass result, it will return an integer which bits indicate whether the pixel belongs to the different classes or not. For example a result of 5 ($0..0101_2$), with single-class means that the output got

classified as belonging to class 5, whereas with multiclass the same result means output got classified as belonging to the class two and zero, as bits 0 and 2 are set to 1.

`int getOutputSize()`
This method simply returns the output size and an integer. This also corresponds to the number of neurons of the last stage of the classifier, i.e., the number of classes.

Learning

These methods provide the possibility to set the expected values (supervised training) and teaching the classifier.

`void setExpected(int classNumber, float expectedValue)`
This method feeds the expected value for each class to the classifier. This means that after computing the output, it should be set the expected value to 0 for all classes except the correct one(s), where the expected value should be set to 1.

`float backpropagate()`
This method backpropagate the error in the same way as for SCAEs, i.e., transmits the error to the previous layer and accumulates parameter modifications.

`float learn()`
This method applies the parameter modifications accumulated by the different `backpropagate` calls.

Utility

The following are general purpose utility methods. This section only covers the basics stuff that should be common among all classifier. However, each classifier shall have an own set of utility methods which are specific to his nature.

`String name()`
Must return a string indicating the name of the classifier (AEClassifier,FFCNN, ...). Useful to avoid using the java "instanceof" everywhere.

`String type()`
Must return a string indicating the type of the classifier (pixel, graph, XML ...). Useful to for easily extend the framework to work with

other data types. At the moment the only possible value implemented is "pixel".

`int getInputWidth()`
Returns the width of the input patch.

`int getInputHeight()`
Returns the height of the input patch.

`void save(final String fName)`
Saves the object on file through the serializable interface.

At this point your classifier is ready to be used.

6.5 Plotting Graphs

Writing in the standard output training errors and classification accuracy is useful for comparing different SCAEs or classifiers. However it might be very enlightening to plot the training error in function of the number of training samples used in order not to compare only the accuracy of different approaches, but also their behavior during the training. The `Tracer` class can be used for this purpose.

As a large quantity of samples can be used during the training (up to millions), plotting all of them it not realistic, nor useful. For this reason, the tracer will sub-sample the data it receives and display only a fraction of it.

Additionally, the plotter can be set not to display anything, and to only store the sub-sampled data into a text file. This can be useful when the framework is used in an environment without graphical user interface, e.g., a cluster, or when users want to make specific plots themselves.

Creating a Tracer

At the creation of the tracer it is necessary to specify the title of the plot, the label of the x and y axis and three additional parameters:

numSamples Specifies what is the expected number of samples which will be fed to the tracer. As the values passed to the tracer will be sub-sampled, it is necessary to know how approximately many of them will be received in order to skip the right amount between two stored values.

maxPoints This is the number of points that will be displayed on the plot after the sub sampling. A value of 200 is suggested.

visible Indicates whether the graphical interface should be used or not.

Listing 6.5: Creating a tracer

```
tracer = new Tracer(
    "Title",
    "x label",
    "y label",
    numSamples,
    maxPoints,
    visible
);
```

Adding a Point

To add point to the tracer, just call the `addPoint(double,double)` method, providing x and y values.

Listing 6.6: Adding a point to the tracer

```
tracer.addPoint(xValue,yValue);
```

Plotting Data

Once all the data has been fed to the tracer, it is possible to plot it to screen as well as saving the plot to a file. In order to have more stable curves, it might be useful to plot the average of the data or some other metric and not just the raw samples. For this reason there are already some built in features, which can be then easily extended by simply adding a methods to the `Tracer` class. The possible trace are:

Raw data Displays the stored samples as they are fed to the tracer.

Cumulated average Displays the average of all samples cumulated along the x axis.

Moving average Displays moving average computed with a moving kernel of 1/10 of `maxPoints`.

Moving median Displays moving median computed with a moving kernel of 1/10 of `maxPoints`.

It is possible to add one or several of them. To do so, use their respective method, as shown below.

Listing 6.7: Adding the tracks on the tracer

```
tracer.addRawData();
tracer.addCumulatedAverage();
tracer.addMovingAverage();
tracer.addMovingMedian();
```

Displaying Data

To visualize the different traces, call the *display()* method.

Listing 6.8: Displaying the plot

```
tracer.display();
```

Saving Data

It is possible to save the plot and data to a file with *savePlot(String fPath)*. There should be no file extension, as two files will be created:

1. A `.tracer` file storing the sub-sampled data, thus allowing users to further analyse it.
2. A `.png` file, storing the plot.

If the `Plotter` was created invisible, the `.png` file will not be created.

Listing 6.9: Saving plot on a file

```
tracer.savePlot(fPath));
```

Bibliography

[1] Warren S McCulloch and Walter Pitts. A logical calculus of the ideas immanent in nervous activity. *The bulletin of mathematical biophysics*, 5(4):115–133, 1943.

[2] Raúl Rojas. *Neural networks: a systematic introduction*. Springer Science & Business Media, 2013.

[3] J. Schmidhuber. Deep learning in neural networks: An overview. *Neural Networks*, 61:85–117, 2015. Published online 2014; based on TR arXiv:1404.7828 [cs.NE].

[4] Kevin J Lang, Alex H Waibel, and Geoffrey E Hinton. A time-delay neural network architecture for isolated word recognition. *Neural networks*, 3(1):23–43, 1990.

[5] Kunihiko Fukushima. Neocognitron: A self-organizing neural network model for a mechanism of pattern recognition unaffected by shift in position. *Biological cybernetics*, 36(4):193–202, 1980.

[6] Takashi Kimoto, Kazuo Asakawa, Morio Yoda, and Masakazu Takeoka. Stock market prediction system with modular neural networks. In *Neural Networks, 1990., 1990 IJCNN International Joint Conference on*, pages 1–6. IEEE, 1990.

[7] Soheil Bahrampour, Naveen Ramakrishnan, Lukas Schott, and Mohak Shah. Comparative study of deep learning software frameworks. *arXiv preprint arXiv:1511.06435*, 2015.

[8] Yangqing Jia, Evan Shelhamer, Jeff Donahue, Sergey Karayev, Jonathan Long, Ross Girshick, Sergio Guadarrama, and Trevor Darrell. Caffe: Convolutional architecture for fast feature embedding. In *Proceedings of the ACM International Conference on Multimedia*, pages 675–678. ACM, 2014.

[9] Frédéric Bastien, Pascal Lamblin, Razvan Pascanu, James Bergstra, Ian Goodfellow, Arnaud Bergeron, Nicolas Bouchard, David Warde-Farley,

and Yoshua Bengio. Theano: new features and speed improvements. *arXiv preprint arXiv:1211.5590*, 2012.

[10] Ronan Collobert, Koray Kavukcuoglu, and Clément Farabet. Torch7: A matlab-like environment for machine learning. In *BigLearn, NIPS Workshop*, number EPFL-CONF-192376, 2011.

[11] Martın Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, et al. Tensorflow: Large-scale machine learning on heterogeneous distributed systems. *arXiv preprint arXiv:1603.04467*, 2016.

[12] Mathias Seuret, Rolf Ingold, and Marcus Liwicki. A Highly-Adaptable Java Library for Document Analysis with Convolutional Autoencoders and Related Architectures. In *Frontiers in Handwriting Recognition (ICFHR), 2016 15th International Conference on*, 2016 (to appear).

[13] Kai Chen, Mathias Seuret, Marcus Liwicki, Jean Hennebert, and Rolf Ingold. Page segmentation of historical document images with convolutional autoencoders. In *Document Analysis and Recognition (ICDAR), 2015 13th International Conference on*, pages 1011–1015. IEEE, 2015.

[14] Mathias Seuret, Kai Chen, Nicole Eichenbergery, Marcus Liwicki, and Rolf Ingold. Gradient-domain degradations for improving historical documents images layout analysis. In *Document Analysis and Recognition (ICDAR), 2015 13th International Conference on*, pages 1006–1010. IEEE, 2015.

[15] Kai Chen, Cheng-Lin Liu, Mathias Seuret, Marcus Liwicki, Jean Hennebert, and Rolf Ingold. Page segmentation for historical document images based on superpixel classification with unsupervised feature learning. In *2016 12th IAPR Workshop on Document Analysis Systems (DAS)*, pages 299–304. IEEE, 2016.

[16] Mathias Seuret, Andreas Fischer, Angelika Garz, Marcus Liwicki, and Rolf Ingold. Clustering historical documents based on the reconstruction error of autoencoders. In *Proceedings of the 3rd International Workshop on Historical Document Imaging and Processing*, pages 85–91. ACM, 2015.

[17] Hao Wei, Mathias Seuret, Kai Chen, Andreas Fischer, Marcus Liwicki, and Rolf Ingold. Selecting autoencoder features for layout analysis of historical documents. In *Proceedings of the 3rd International Workshop on Historical Document Imaging and Processing*, pages 55–62. ACM, 2015.

[18] Oussama Zayene, Mathias Seuret, Sameh M Touj, Jean Hennebert, Rolf Ingold, and Najoua E Ben Amara. Text detection in arabic news video based on swt operator and convolutional auto-encoders. In *2016 12th IAPR Workshop on Document Analysis Systems (DAS)*, pages 13–18. IEEE, 2016.

[19] Mathias Seuret, Marcus Liwicki, and Rolf Ingold. Focus on the Center – Multilevel Features from Convolutional Autoencoders for Document-Part Classification Tasks. *submitted*.

[20] A. Garz, M. Seuret, F. Simistira, A. Fischer, and R Ingold. Creating ground truth for historical manuscripts with document graphs and scribbling interaction. In *Proc. 12th Int. Workshop on Document Analysis Systems*, pages 126–131, 2016.

[21] Ian Goodfellow Yoshua Bengio and Aaron Courville. Deep learning. Book in preparation for MIT Press, 2016.

[22] Marcel Würsch, Rolf Ingold, and Marcus Liwicki. Sdk reinvented: Document image analysis methods as restful web services. In *Document Analysis Systems (DAS), 2016 12th IAPR International Workshop on*, pages 90–95. IEEE, 2016.

[23] Hrushikesh Narhar Mhaskar and Charles A Micchelli. How to choose an activation function. *Advances in Neural Information Processing Systems*, pages 319–319, 1994.