Alma Mater Studiorum · Università di Bologna

SCUOLA DI SCIENZE

Corso di Laurea Magistrale in Informatica

Advanced industrial OCR using Autoencoders

Relatore:
Chiar.mo Prof.
Luigi Di Stefano

Correlatore:
PhD.
Luis Goncalves

Presentata da:
Mauro Belgiovine

Sessione I
Anno Accademico 2016/17

"Life is like riding a bicycle.
To keep your balance, you must keep moving."
Albert Einstein

Introduction

The content of this thesis describes the work done during a six-month internship at Datalogic ADC, whose offices are located in Pasadena (CA). The aim of my research was to use a specific type of neural network, called Autoencoder, for character recognition or validation purposes in an industrial OCR system, contributing to the development and enhancement of Datalogic's software products employed in this field.

This type of network is particularly interesting as it allows to learn a reduced dimensional encoding of a certain data collection, such as a dictionary of symbols. Industrial OCR systems are used in real-time scenarios such as assembly or quality control chains and it presents stringent requirements for robustness to input variability (e.g. character distortions, variable lighting, incorrect positioning of objects) and processing times. In contrast to the classic pattern matching-based character recognition methods, which in these cases should pre-process and compare inputs with vast amount of templates, the use of a neural network would allow us to not only recognize characters, but also their possible variations in a limited amount of memory, corresponding to the network's parameters itself.

After a first phase of studying and testing the model on MNIST (a classic handwritten digt dataset used in literature as a benchmark), our work focused on using Autoencoder in two different areas. First, a *classifier* based on Denoising Autoencoder was created, initializing network parameters in *unsupervised* manner and using them as a starting point to train a character classifier with a *supervised* approach. Subsequently, we studied a method

to use Autoencoders as a second level classifier, or *verifier*, which can be plugged-in any OCR (or other visual recognition tasks) pipeline to better distinguish false activation from the correct ones, under conditions of uncertainty at the primary classifier level due to the high variability of input data.

Both architectures were evaluated on some real Datalogic's customer datasets and experimental results are presented in this work.

Contents

Introduction **i**

1 Deep Learning **1**

1.1 Artificial neural networks 2
1.1.1 The perceptron 2
1.1.2 Layer-wise structure 4
1.1.3 Feed-forward step 5
1.1.4 Optimization 5
1.1.5 Representational power 8
1.2 Autoencoders 8
1.2.1 Single layer Autoencoder 8
1.2.2 Denoising Autoencoders 10
1.2.3 Stacked autoencoders 13
1.3 Experiments with (S)DAE 16
1.3.1 Increasing noise in DAE 18
1.3.2 Pre-training only vs. fine-tuning 18

2 Autoencoders for digit classification **21**

2.1 Background and state-of-the-art solutions 22
2.1.1 Computer Vision perspective 22
2.1.2 Deep Learning perspective 22
2.2 Autoencoder-based OCR 23
2.3 Experimental setup 24
2.3.1 MNIST classifier 24

2.3.2 A real-world case 27
2.3.3 Hyper-parameters optimization 31
2.4 Testing on real images 32
2.4.1 From fully connected to convolutional layers 33
2.4.2 Classifier evaluation 33
2.5 Experimental setup 36
2.5.1 Basic SDAE-C performance 36
2.5.2 Increasing SDAE-C performance 38
2.5.3 SDAE-C final considerations 41
2.5.4 Additional Autoencoder to deal with FPs 42
2.5.5 Improved SDAE-C performance 43
2.6 Performance on complete folders 44
2.7 Conclusions 47

3 Autoencoders as second level classifier 49
3.1 Training a SDAE for each class 49
3.2 Define the reconstruction threshold 50
3.2.1 Modeling reconstruction error with a Normal distribution 51
3.2.2 Verification dataset to define threshold 51
3.3 Using AEVs to verify classifier outputs 52
3.4 Experimental setup 54
3.4.1 Visualize reconstruction errors on a real image 54
3.4.2 Shallow vs. deep architecture 55
3.4.3 Practical application of AEVs 56
3.4.4 From multiple fully-connected reconstructions to a single 2D convolution 58
3.5 Integration with HOG-OCR 59
3.5.1 Final results 60
3.5.2 Conclusions 64

4 Future directions 65
4.1 Stacked Denoising Autoencoder Classifier (SDAE-C) 65

4.2 Autoencoder Verifiers (AEV) 66

Bibliografia **67**

List of Figures

1.1 Graphic representation of a biological neuron and its mathematical counterpart, the perceptron. 2

1.2 Example of a 3-layer neural network structure. 4

1.3 SGD performed over a cost function with 1 parameter. 6

1.4 Structure of a single layer Autoencoder. 9

1.5 A graphical explanation of the training method for Denoising Autoencoders. Here the encoding function is denoted with f_θ and the decoding function with $g_{\theta'}$. 11

1.6 Manifold learning perspective of Denoising Autoencoders. Stochastic operator $p(X|\tilde{X})$ is used to "project back" on the manifold the corrupted samples $\tilde{\mathbf{x}}$. 13

1.7 General architecture of a deep Autoencoder. 14

1.8 Stacking denoising autoencoders. After training a first level denoising autoencoder, its learnt encoding function f_θ is used on clean input (left). The resulting representation is used to train a second level denoising autoencoder (middle) to learn a second level encoding function $f_\theta^{(2)}$. The procedure is iterated on all successive levels. 15

1.9 Training without (left) and with Gaussian noise (right) effect on features learned (starting from the same weights initialization). 17

1.10 Average RMS per pixel over the MNIST dataset while training with increasing noise. 17

1.11 SDAE architecture trained using MNIST. The number below each layer indicates the number of neurons. 19
1.12 Reconstructions for 5 digits from the MNIST dataset. Images on the left are from pre-training only strategy and the ones on the right uses both pre-training and fine-tuning. At each image, columns from left to right are: original input, reconstruction obtained using Lev. 1, Lev. 2, Lev. 3, Lev. 4 encoding layer. 19

2.1 Modified architecture of SDAE used to build a MNIST classifier 24
2.2 Classification accuracy on the MNIST Test dataset using a SDAE-based classifier. Comparison between a classic *pre-training* strategy (top) and *fine-tune* only strategy (bottom). . 26
2.3 Abstract model of pictures in the dataset. 28
2.4 Digit samples from the synthetic dataset. 29
2.5 Background class samples. 30
2.6 Graphical explanation of Fully Connected (FC) to Convolutional (CONV) layer conversion for a 2-layer network. 32
2.7 Comparison between features learned at first layer of the SDAE-C using $\sigma = 0.2$ (left) and $\sigma = 0.4$ (right). Using more noise during training let the feature be more general but more noisy as well, which makes them fire the neurons even with fewer constraints on the input features, but also increases the chance of wrong activations. 37
2.8 Weights visualization of 200 neurons at the first layer of the SDAE-C (trained using noise level $\sigma = 0.2$). 39
2.9 Activation maps for each label obtained with a SDAE-C trained at noise level $\sigma = 0.4$. The images are referred to labels in this order (left to right, top to bottom): `0`, `1`, `2`, `3`, `5`, `6`, `7`, `8`, `9`, `dot`, `background`. Last figure is left only as a reference for activations positions. 40

2.10 ROC curve to show TPs rate over average FPs per image at dataset's folder level. Starting from first, each successive point on the curve indicates an increment of 0.5 on the threshold used to discard FPs. 46

3.1 An example of OCR pipeline with the additional verification step, performed by Autoencoder Verifiers on the classifier output. 52

3.2 Difference in RMS error between a correctly labeled sample and one that we would like to reject. The different outcomes using two AEVs are shown for each sample. 53

3.3 Visualization of the reconstruction error obtained using a AEV trained on class "0" at each location in a real input image. . . 55

3.4 The green rectangle is the original crop, generated from the detection coordinates. The red rectangle is the area around the original detection, given $f = 0.5$. Each pixel in this area will result in a new crop to evaluate. The whole image is the total area extracted from the input that will be processed. . . 57

3.5 Encoding (above) and decoding (bottom) steps performed over the wider crop area using a convolutional approach. 58

3.6 Recognition rate using HOG-OCR on *Snacks* dataset with and without AEVs application on detections. Results are referred to line level accuracy (above) and at image level (bottom). . . 61

3.7 Overal recognition rate using HOG-OCR on *Snacks*, *Markers* and *Flour* complete datasets with and without AEVs application on detections. Results are referred to image level (line level is also reported for *Snacks*). 62

Chapter 1

Deep Learning

Nowadays, companies are turning to Deep Learning to solve hard problems, like object or face recognition in images, object tracking, pose prediction, image captioning, speech recognition, natural language processing and many more. Deep Learning is essentially a sub field of Machine Learning that studies statistical models called *deep neural networks*. These models are able to learn complex and hierarchical representations from raw data, unlike conventional, hand crafted Machine Learning models used for features extraction.

Researchers have been studying Deep Learning since 1940-1960s, when the first biologically inspired models, such as the Perceptron, have been proposed. In 1960-1980s a second wave of research started, fueled by the discovery of backpropagation algorithm, which is still used today as one of the principal components and fundamental steps to actually let artificial neural networks learn. A notable contribution has been the invention of Convolutional Neural Networks (CNN), moving away for the first time from the classic fully-connected models and introducing a new notion of "local" or "spatial" feature extraction (as opposed to the general ones learned by fully-connected models) that has been particularly successful with image and natural language processing. In 2006 newer and more articulated neural network models have been proposed, such as recurrent neural network and

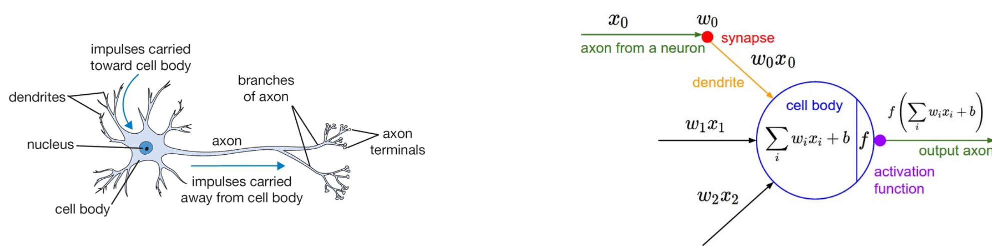

Figure 1.1: Graphic representation of a biological neuron and its mathematical counterpart, the perceptron.

deep beliefs network, starting what is considered to be the modern era of Deep Learning.

As today, Deep Learning is gaining more and more attention, winning challenges even beyond its conventional applications area. Its success and progress have been made possible by the increase of computational resources, especially with the adoption of GPGPU to address the massively parallel computations involved with large neural networks and big datasets, the increase of available annotated data and the community-based involvement to open source codes and frameworks used to build and share models among people working in this field.

1.1 Artificial neural networks

Artificial neural networks are inspired by the structure of the cerebral cortex, although they're not comparable with the complexity of such biological structures. In this section we will cover some of the principles upon most of neural network models are built on.

1.1.1 The perceptron

The *perceptron* [1] is the building block of an artificial neural network. It is a mathematical representation of a simplistic biological neuron, which receives input signals from its dendrites and produces output signal along its axon. In the computational model of a neuron, the signals that travel along

the axons (e.g. x_0) interact multiplicatively (e.g. w_0x_0) with the dendrites of the other neuron based on the synaptic strength at that synapse (e.g. w_0). See Fig. 1.1 for reference.

The idea is that the synaptic strengths (the weights w) are learnable and control the strength of influence (and its direction: excitory/positive weight or inhibitory/negative weight) of one neuron on another. In the basic model, the dendrites carry the signal to the cell body where they all get summed. Thus, a perceptron k is essentially a unit performing a linear mapping between the input x and a template w, adjusting the output with a bias value b_k:

$$y_k = \sum_i w_i x_i + b_k$$

If we define a linear activation function, for example:

$$g(y_k) = \begin{cases} 1 & \text{if } y_k > 0 \\ 0 & \text{else} \end{cases}$$

we could build and train a single perceptron to obtain a linear classifier that distinguish any input x based on its linear mapping y_k value.

The difference in artificial neural networks is that the neuron can *fire* if the final sum is above a certain threshold, sending a spike along its axon based on the intensity of y. We model the firing rate of the neuron with an activation function f, typically a non-linear function which output represents the frequency of the spikes along the axon. Historically, a common choice of activation function is the sigmoid function σ, since it takes a real-valued input (the signal strength after the sum) and squashes it to range between 0 and 1, but different and better activation functions have been studied and widely adopted for this purpose, such as *hyperbolic tangent* (or *tanh*) and rectifier functions, *ReLU* and *Softplus*. Such non-linearity applied to the linear mapping is what makes them different from a simple linear classifier (which has been the typical application for perceptrons in Machine Learning) and what makes neural networks so powerful when it comes to approximating complex functions.

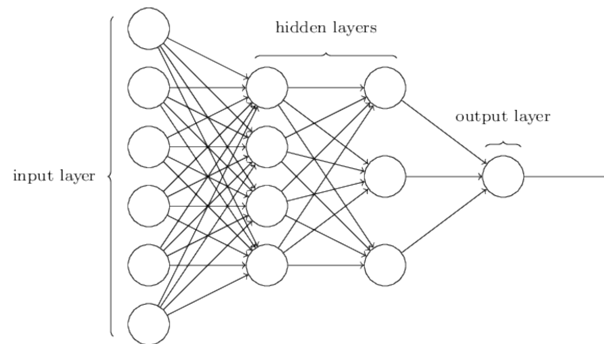

Figure 1.2: Example of a 3-layer neural network structure.

1.1.2 Layer-wise structure

Like in the cerebral cortex, there can be several layers of interconnected perceptrons. Neural networks are modeled as collections of neurons connected in an acyclic graph, so that the outputs of some neurons can become inputs to other neurons. These models are usually organized in *layers* of neurons: a regular neural network will present 1 or more *fully-connected layers* in which neurons of two adjacent layers are fully pairwise connected, but with no connection among neurons of the same layer. A simple neural network is shown in Fig. 1.2.

For naming conventions, we refer to the first layer as the *input layer*, where each neurons represents a single feature of the input data that we want to process. As an example, if we built a neural network to process gray-scale images, each of these neurons will host a single pixel value of the image, expressed as the pixel intensity (e.g. a real value between 0 and 1). The layers in the middle are called *hidden layers* and each of the neurons in these layers is trained to learn a *template*, a particular configuration of the input features coming from the previous layer. Each subsequent hidden layer is used to learn a hierarchical, higher level representation of the templates learned in the previous one. In the end, we have a *output layer*. It is worth to note that, unlike all layers, the output layer neurons typically don't use an activation function but a linear identity activation function, because they're

typically used to represent class scores for classification tasks (i.e. arbitrary real values) or some kind of real-valued target for regression tasks.

1.1.3 Feed-forward step

We refer to a *feed-forward computation* as the processing of the input data through all the hidden layers and the output layer. If we imagine an artificial neural network as a black box, a feed-forward step would simply involve giving an input to the neural network and receiving an output from it.

One of the primary reasons for the layer-wise organization is that it makes evaluating inputs very simple and efficient by combining all the dot products and additions happening at each layer in matrix-vector operations. Usually, all the weights in a layer are stored in a single matrix W having size $N \times M$, where N is the number of input values (i.e. the number of outputs/neurons in the previous layer) and M is the number of neurons in this layer (i.e. the number of templates this layer wants to learn). Therefore, the forward pass of a single fully-connected layer will correspond to one vector-matrix multiplication (considering a single input) followed by a bias offset and an activation function, that is:

$$\mathbf{y} = f(\mathbf{W}\mathbf{x} + \mathbf{b})$$

By applying the forward step to each pair of layers in the network, using the output from previous layer as input for the subsequent one, we will obtain the final output of our network.

1.1.4 Optimization

In order to correctly train a network to perform some specialized task on the input and give us meaningful output, we need to define a *loss function* $\mathcal{L}$, a function used to quantify how “similar” is the output of the neural network with respect to our expectations. This function can be defined in

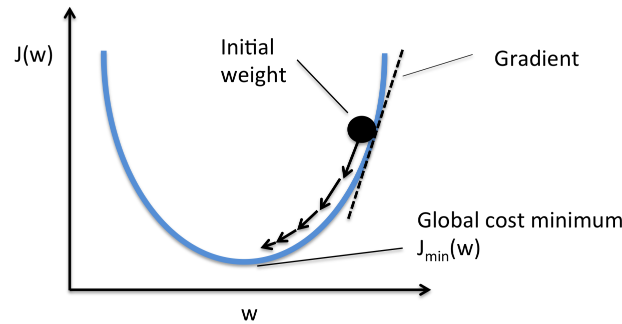

Figure 1.3: SGD performed over a cost function with 1 parameter.

different ways and it strongly depends on what kind of computation the neural network will address. As an example, for image classification tasks the network will get an image as input and will output a score for each class, based on the features extracted from that particular image: in this case, we need a loss function that tells us how correct is the score assigned to each class with respect to the one we expect for that input. The higher the value of this function, the higher the loss.

After being able to evaluate the performance of our neural network, we can now effectively train it on a dataset, which essentially translates into adjusting weights and biases at each layer in order to obtain the desired output. The loss function lets us quantify the quality of any particular set of weights W at each layer, so the goal of training is to find a configuration of those weights that minimizes the loss function, hence solving an optimization problem.

Gradient descent with backpropagation

Adjusting the weights means finding a direction in the multi-dimensional weights parameter space that would likely bring us to a weights configuration that corresponds to a minimum for the loss function $\mathcal{L}$. Using a method called *Stocastic Gradient Descent* (SGD), we can compute the *best* direction along which we should change our weights. By applying a small *learning step* to the actual weights configuration (e.g. incrementing them by a small value), we

can test what is mathematically guaranteed to be the direction of the steepest descent, which is related to the *gradient* direction of $\mathcal{L}$ between the actual configuration and the one we want to test. To compute this gradient, we will need to compute the contribution to the value of $\mathcal{L}$ given by all the weights and biases in our network, essentially by computing all the *partial derivatives* $\frac{\delta\mathcal{L}}{\delta w}$ and $\frac{\delta\mathcal{L}}{\delta b}$ of every single w and b. Once the gradient for $\mathcal{L}$ is determined, the weights are modified in the negative direction of its value, hopefully making our loss function decrease. By iterating this operation several times over the complete dataset, even if there's no guarantee of convergence, SGD will eventually lead us to a weights configuration that has minimum (either global or local) cost for the loss function. A graphical explanation of this concept in 1D is shown in 1.3.

Computing $\mathcal{L}$ gradient is a highly expensive operation, especially if we consider that a neural network could have up to millions of parameters, and special algorithms have been developed to make it both efficient and analytically correct. The widely adopted algorithm to compute gradients is called *backpropagation* and it computes local gradients at each operational gate in the network and then performs a backward pass, starting from the last gate, that recursively applies the so called *chain rule* to combine the local gradients all the way up to the input layer. A detailed explanation of backpropagation can be found in [11].

Also, several gradient descent algorithms have been proposed and an overview can be found at [10]. Choosing one algorithm or another leads to different outcomes and performances for the optimization problem. Moreover, *learning step* hyper-parameter is crucial to correctly search for the minimum in the parameter space: a learning step too small will make the algorithm converge slower, while a learning step too big could let us never reach a minimum.

1.1.5 Representational power

Neural networks can be trained to model very complex functions. The *Universal approximation theorem* [12] states that given any continuous function $f(x)$ and some $\epsilon > 0$, there exists a neural network $g(x)$ with one hidden layer (with a reasonable choice of non-linearity) such that

$$\forall x, |f(x) - g(x)| < \epsilon$$

. As a result of this theorem, we can assume that any neural network with at least one hidden layer can approximate any continuous function. In practice, though, it has been shown empirically that a neural network with more hidden layers typically works better than one with a single hidden layer, approximating complex functions even better and in a hierarchical way that better fits real-world problems.

1.2 Autoencoders

An Autoencoder (AE) is a particular model of artificial neural network used to learn an *efficent coding*. This architecture aims to find a “compact” representation for a set of data in an *unsupervised* manner (i.e. using unlabeled data), with the main purpose of performing dimensionality reduction of such datasets.

1.2.1 Single layer Autoencoder

Architecturally, the simplest form of an Autoencoder is a feed-forward, non-recurrent neural network having an input layer, an output layer and a single fully-connected hidden layer that will learn the *code*. Unlike a common neural network, the output layer has the same number of nodes as the input layer and the encoding is learned at the hidden layer by training the network to reconstruct (almost) perfectly their original input. The hidden layer has

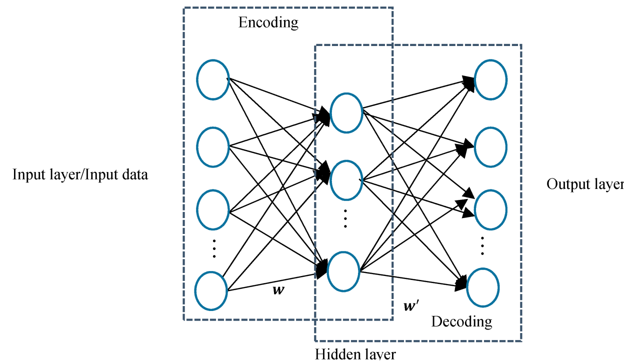

Figure 1.4: Structure of a single layer Autoencoder.

typically less neurons than the input/output[1], so that dimensionality of data is reduced and the AE learns interesting features (otherwise, it could end up learning the identity function, which doesn't generalize well).

An Autoencoder can be decomposed in two main parts:

- **encoder**: the set of layers used to produce the *encoding* of the input data, i.e. the input layer and the hidden layer;
- **decoder**: the set of layers used to decode the input back to its original dimensionality, i.e. the hidden layer and the output layer.

The encoding step can be formalized mathematically in this way:

$$\mathbf{y} = f(\mathbf{W}\mathbf{x} + \mathbf{b})$$

which is basically the forward pass of the input through the hidden layer used to produce the encoding $\mathbf{y}$. The decoding step is the following:

$$\mathbf{z} = f(\mathbf{W}'\mathbf{y} + \mathbf{b}')$$

where $\mathbf{z}$ is the reconstruction of the original input $\mathbf{x}$, while $\mathbf{W}'$ and $\mathbf{b}'$ are respectively the weights and biases used for the decoding step using the

[1]In case the number of neurons at the hidden layer is greater than the input/output layers, it will be referred as a *sparse* Autoencoder.

encoding $\mathbf{y}$ as input. Thus, we have two different sets of weights and biases for the hidden layer, one for the encoding phase and the other for the decoding phase. In order to obtain a reconstruction from these two forward steps, $\mathbf{W}$, $\mathbf{b}$, $\mathbf{W}'$ and $\mathbf{b}'$ should be optimized such that the *reconstruction error is minimized*. This lead us to define the loss function as the squared error of the reconstruction:

$$\mathcal{L}(\mathbf{x}, \mathbf{z}) = \|\mathbf{x} - \mathbf{z}\|^2$$

so that, while training the network, the weights will be adjusted in order to minimize the AE's reconstruction error over a given dataset. Here we are simply measuring the squared distance between the input and the reconstruction vectors, but other cost function can be used, depending on the dataset and the application.

Since the coding learned is a lossy compression of the datasets, it can be seen as good compression for the training samples, and hopefully for other similar inputs as well, but not for arbitrary inputs. That is the sense in which an AE generalizes: it gives low reconstruction error on input samples having a distribution of values similar to the one in the training dataset, but high reconstruction error on samples unseen during the training.

1.2.2 Denoising Autoencoders

Authors in [6] says that in order to force the hidden layer to learn more robust features and avoid learning the identity function, we can train the AE to reconstruct the input from a *corrupted version* of it.

The Denoising Autoencoder (DAE) is a *stochastic* version of the Autoencoder. Intuitively, a Denoising Autoencoder does two things: it encodes the input and undo the effect of a corruption process stochastically applied to the input. A good encoding is expected to capture stable structures in the form of dependencies and regularities characteristic of the (unknown) distribution of its observed input. For high dimensional redundant input (such as images), such structure are likely to depend on evidence gathered from

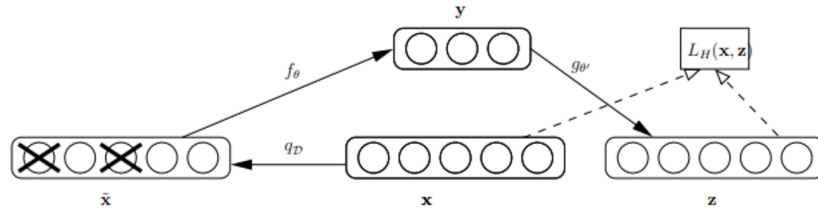

Figure 1.5: A graphical explanation of the training method for Denoising Autoencoders. Here the encoding function is denoted with f_θ and the decoding function with $g_{\theta'}$.

a combination of many input dimensions: they should be recoverable from partial observations only.

Training with noise

If we apply noise to $\mathbf{x}$ with a stochastic mapping, we will obtain a corrupted version $\tilde{\mathbf{x}} \sim q_\mathcal{D}(\tilde{\mathbf{x}}|\mathbf{x})$. Using it as a training example forces the AE to learn a more clever mapping: if we keep minimizing the cost function $\mathcal{L}(\mathbf{x}, \mathbf{z})$ (i.e. on the original version of the input) the AE will learn a code able to remove the stochastic noise, leading to a more robust encoding.

Although the kind and amount of noise to be applied should be accurately chosen with respect to the value distribution of the dataset we are interested in, some general corruption methods are:

- Additive isotropic Gaussian noise: $\mathbf{x}$ values are altered according to a normal distribution $\mathcal{N}(\mu, \sigma^2)$ having $\mu = 0$;
- Zero-masking noise: a fraction v of the elements of $\mathbf{x}$ (chosen at random for each sample) is forced to 0;
- Salt-n-pepper noise: a fraction v of the elements of $\mathbf{x}$ (chosen at random for each sample) is set to their minimum or maximum possible value, according to a fair coin flip.

Especially if we are interested in a classification objective, training a neural network with noisy input (i.e. jitter) enhances generalization of the tem-

plates learned, because it act as an augmentation of the original training set with additional, stochastic distorted samples. Together with some other kind of data preparation, like a ad-hoc augmentation (e.g. stretch, rotate or scale image samples), this method can help the network to learn an encoding of the input dataset that better fits its distribution of data and can be also used to initialize a classifier network for supervised learning. In fact, initializing weights in a greedy, layer-wise manner with an unsupervised learning method and then refining them during the supervised training has been shown to yeld significantly better local minima than random initialization of deep neural networks [13] and achieve better generalization [14].

Motivations for a denoising approach

The *denoising* approach is advocated and investigated as a training criterion to learn useful features that will constitute a more robust representation of data. As we will see later, corruption and denoising procedure is applied not only to the original input, but also recursively to intermediate representation while training deep Autoencoders.

One intuitive, geometric interpretation of this approach is given under the *manifold assumption*, which states that natural high-dimensional data concentrates close to a non-linear low-dimensional manifold (which is the same assumption used for PCA). While training with a denoising approach, we are learning a *stochastic operator* $p(X|\tilde{X})$ which maps a corrupted $\tilde{X}$ to its original form X. Corrupted examples are more likely to be outside and far from the manifold than the uncorrupted ones. Thus, $p(X|\tilde{X})$ learns a mapping that tends to go from lower probability points $\tilde{X}$ to high probability points X. Moreover, when the noise is large $\tilde{X}$ is farther from the manifold and $p(X|\tilde{X})$ should learn to project back to it also very far points.

The Denoising Autoencoder can be seen as a way to define and learn a manifold, having its encoding $f_\theta(X) = Y$ as a mapping of the input to a point on the coordinate system of the lower-dimensional manifold. More interpretations of how Denoising Autoencoders work are given in [6].

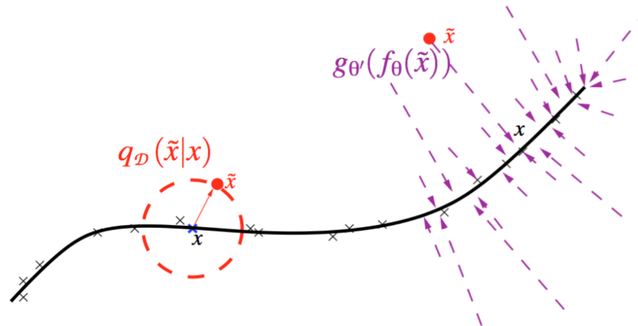

Figure 1.6: Manifold learning perspective of Denoising Autoencoders. Stochastic operator $p(X|\tilde{X})$ is used to "project back" on the manifold the corrupted samples $\tilde{\mathbf{x}}$.

1.2.3 Stacked autoencoders

Deep neural networks use a cascade of layers of nonlinear processing units, instead of a single hidden layer. Each layer uses the output from the previous layer as input. In this way, the nets learn multiple levels of representations that correspond to different levels of abstraction: the levels form a hierarchy of concepts automatically learned from the dataset.

Guiding the training of intermediate levels of representation using unsupervised learning, which can be performed locally at each level, has been one of the major breakthrough in the deep learning community: using this method, it has been possible for the first time[2] to train efficiently deep architectures (e.g. more than 2 or 3 layers) that outperformed their respective shallow counterpart in many different applications. Algorithms using stacked Restricted Boltzman Machine (called Deep Belief Networks) and Autoencoders for this purpose have been introduced in [4] and [5].

More specifically, Autoencoders can be *stacked* in a greedy, layer-wise fashion in order to create a hierarchical coding of features extracted from a

[2]Except for neural networks extracting spatial features, as Convolutional Neural Networks.

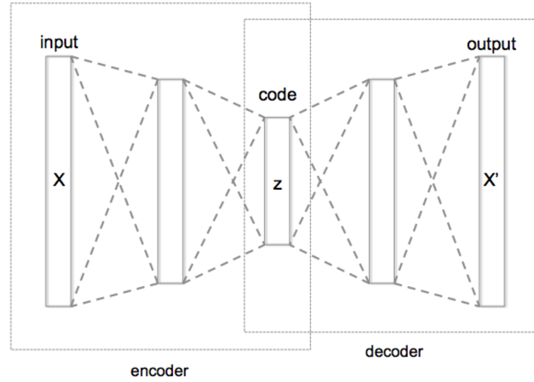

Figure 1.7: General architecture of a deep Autoencoder.

particular dataset. These features can then be used to initialize the weights of a deep neural network for supervised learning tasks (e.g. classification).

Architecture

A Stacked (or deep) Autoencoder (SAE) it's built using progressively smaller AEs, each of them having an input/output size equal to the hidden layer size of the previous one. The encoding at a certain layer is used as the input of the next autoencoder, until we reach the smallest encoding layer. In the same way, the decoding phase starts from the innermost layer and its reconstruction is used as the input of the next decoding step, until we reach the original dimensionality of data. Fig. 1.7 shows a high level representation of this model.

Formally, for n SAEs, indexed from 1 to n, if we denote the encoding function at AE_i (i.e. the i-th Autoencoder) with $h_i(\mathbf{x}) = \mathbf{y}$ and the reconstruction as $r_i(\mathbf{y})$, the encoding of the original input $\mathbf{x}$ throughout the network at AE_i will be:

$$\mathbf{e}_i = h_i(h_{i-1}(...h_2(h_1(\mathbf{x}))...))$$

while its reconstruction will be:

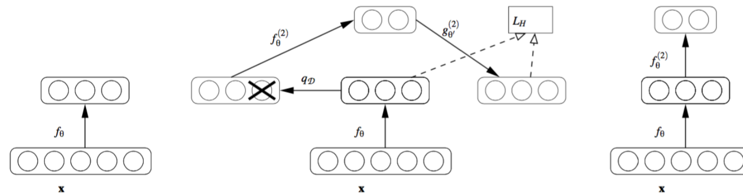

Figure 1.8: Stacking denoising autoencoders. After training a first level denoising autoencoder, its learnt encoding function f_θ is used on clean input (left). The resulting representation is used to train a second level denoising autoencoder (middle) to learn a second level encoding function $f_\theta^{(2)}$. The procedure is iterated on all successive levels.

$$\mathbf{y} = r_1(r_2(...r_{i-1}(r_i(\mathbf{e}_i))...))$$

Pre-Training

Training this architecture in a greedy, layer-wise fashion means essentially training the network one AE at a time. It is commonly referred to as *pre-training* for the weights initialization purpose explained above.

Starting from the first layer, we train the hidden layer to find an efficient coding of the dataset, minimizing the reconstruction error over the original inputs. For the next AE, we will use the same dataset but with $h_1(\mathbf{x})$ as the input, thus learning a coding for the encoding of such data coming from the previous layer and minimizing the reconstruction error of the encodings. The procedure is iterated until the innermost coding has been learned.

The training process is the very same for Stacked Denoising Autoencoders (SDAE), with the only difference that corruption is applied on the input (either the original one or an encoding from a previous level) while training each AE. See Fig. 1.8 for a graphical explanation.

Fine-tuning

The *fine-tuning* training strategy consists in updating the weights of a Stacked Autoencoder all at once at each training epoch. After computing a complete forward pass in the network, the reconstruction error is back propagated through all the layers and all the weights are updated at once, as if they belong to a single model, in order to minimize the reconstruction error.

This strategy is commonly found in Deep Learning and, coupled with pre-traing, it can be used to greatly improve the performance of a Stacked Autoencoder.

1.3 Experiments with (S)DAE

The neural network implementations presented throughout this work are built using TensorFlow[3] and they have the following common features:

- The non-linear activation chosen for neurons is a rectifier, specifically a smooth approximation of the ReLU function called *Softplus* [23];
- The weights are initialized using the Xavier method [18];
- The gradient descent strategy chosen is Adam optimizer [20];
- At each training epoch, the dataset is processed in mini-batches of 128 images at a time.

Other hyper-parameters such as number of training epochs, number of layers and neurons per layer, learning rate for gradient descent algorithm and the amount and type of noise applied are chosen accordingly to the experiments requirements and goal.

[3]TensorFlow is an open source software library developed by Google and intended for building Machine Learning and Deep Learning models.

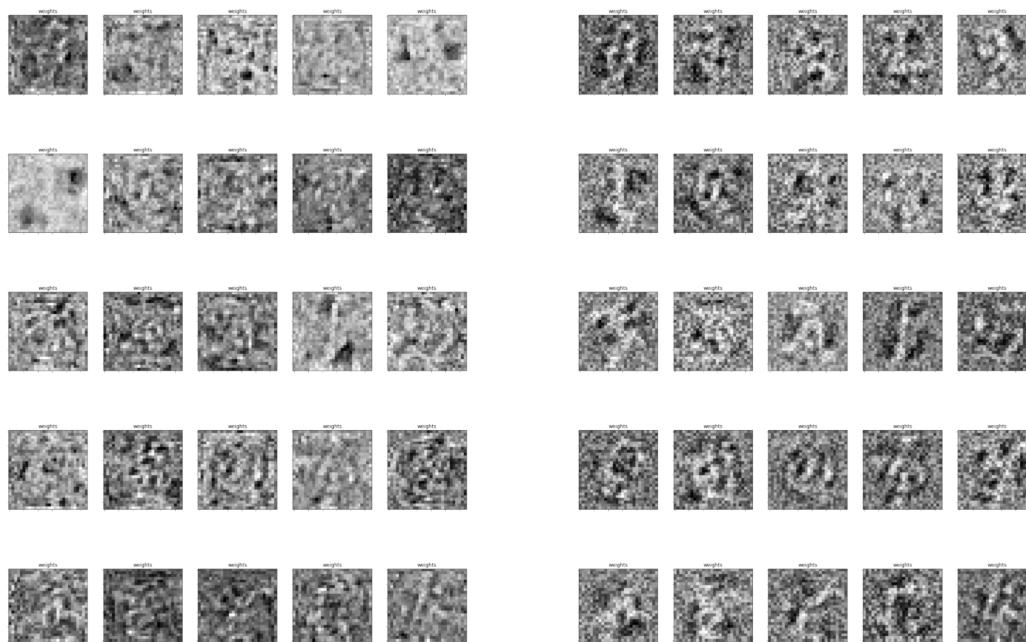

Figure 1.9: Training without (left) and with Gaussian noise (right) effect on features learned (starting from the same weights initialization).

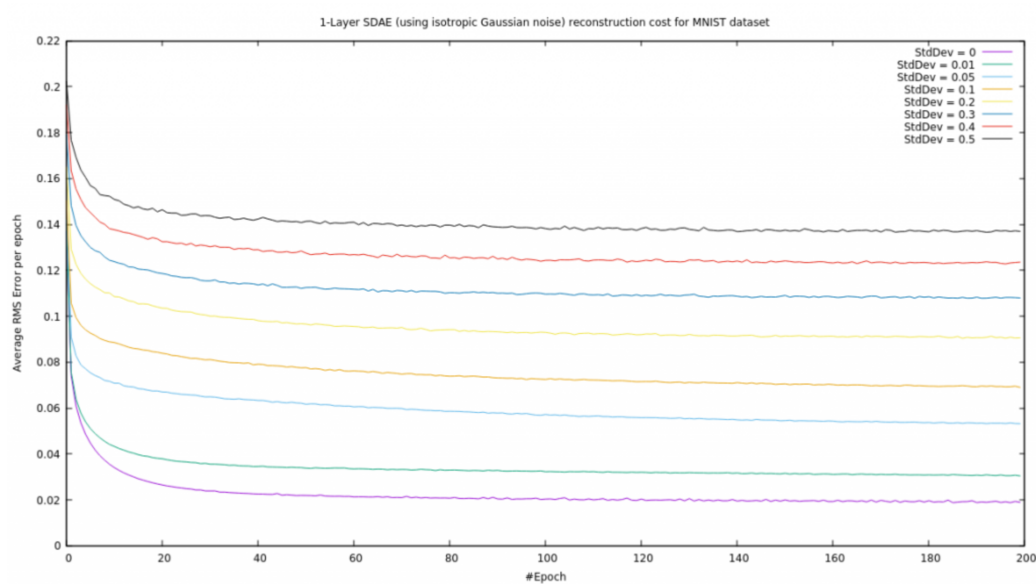

Figure 1.10: Average RMS per pixel over the MNIST dataset while training with increasing noise.

1.3.1 Increasing noise in DAE

In order to look at how the features learned change with different level of corruption applied at the input, as a first experiment we trained a single layer DAE on the MNIST dataset using Gaussian noise, increasing the noise level at each experiment.

For this experiment we used a 500 neurons hidden layer, trained for 50 epochs at a learning rate of 0.0001. As Fig. 1.9 shows, features of the MNIST dataset learned at each neuron seems to become sharper as noise increases. While the templates learned seems to be more robust, increasing noise tends also to give a higher average reconstruction error on the training samples (here expressed as RMS per pixel at each training epoch), as depicted in Fig. 1.10.

1.3.2 Pre-training only vs. fine-tuning

In this experiment we wanted to investigate the effect of different training strategies on a SDAE, with or without the application of fine-tuning phase in the training strategy.

We implemented the SDAE architecture shown in Fig. 1.11 and trained it over the MNIST dataset. The number of neurons at each layer and the number of layers are chosen based on the related work presented in [3]. At first, we looked at how the input reconstruction looks like at each SDAE level before and after the fine-tuning phase. The hyper-parameters used for this test are the following:

- 30 training epochs for both pre-training at each level and fine-tuning;
- learning rate was set to 0.001;
- in this case, no corruption has been applied while training[4].

[4] For visualization purpose, training with noise has the only effect of introducing more noise in the reconstructions.

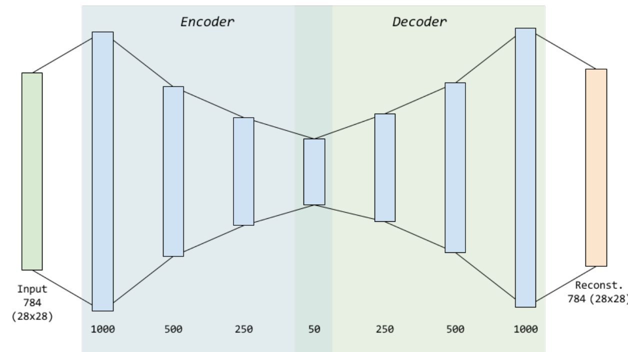

Figure 1.11: SDAE architecture trained using MNIST. The number below each layer indicates the number of neurons.

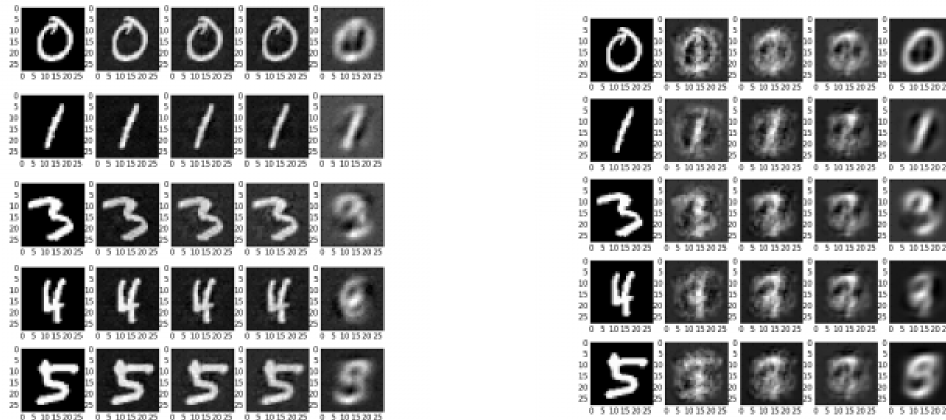

Figure 1.12: Reconstructions for 5 digits from the MNIST dataset. Images on the left are from pre-training only strategy and the ones on the right uses both pre-training and fine-tuning. At each image, columns from left to right are: original input, reconstruction obtained using Lev. 1, Lev. 2, Lev. 3, Lev. 4 encoding layer.

The first (left) set of reconstructions in Fig. 1.12 shows essentially what is the effect of pre-training on the SAE: the higher the number of neurons (as at the outermost layers), the better the reconstruction obtained because of a better approximation of the input in the learned code. Using the first layer encoding only (i.e. the largest in terms of neurons), we obtain a near perfect reconstruction of the input. As the input is encoded down to lower dimensionality (i.e. being processed by the innermost layers), the reconstruction becomes more and more different from the input, but more general as well: this reflects how the hierarchical, low dimensional code learned by the Autoencoder embrace all the variability of data for a specific set of samples in the whole dataset.

The second (right) set of reconstructions shows how the weights are modified by the fine-tuning phase, leading to a poorer reconstruction at outermost layers, due to all-at-once weights adjustments introduced, but improving consistently the reconstructions obtained with a complete forward pass of the neural network.

Chapter 2

Autoencoders for digit classification

Optical character recognition (OCR) aims to convert images of typed, handwritten or printed text into machine-encoded text. At industrial level, a very common OCR application is to read printed labels on products, usually employing a fixed camera that takes pictures at a high rate of such products while moving on conveyor belts. The automaton of this process is critical, in order to make tasks as product dispatching or product checking more efficient. The key ideas behind the design of such systems is the reading throughput (i.e. how many picture per second it can read) and its reliability (i.e. how accurately it can recognize characters). Moreover, systems typically need to be trained only for a little amount of time and using as few samples as possible, while keeping a robust detection over unpredictable condition and variations of characters, due to mispositioned, out of focus products, light variation or degradation of printing quality over time.

2.1 Background and state-of-the-art solutions

2.1.1 Computer Vision perspective

Classical approaches for character recognition involve hand-built and highly-tuned modules to perform some form of template matching on the images taken by the camera, either comparing single font templates or certain features extracted from the input. Depending on the application, these modules perform three main processing steps:

- Pre-processing and segmentation;
- Recognition process;
- Post-processing of acquired data.

Pre-processing techniques, such as *de-skewing*, *binarization*, *aspect ratio normalization* and *scaling*, are used to remove noise and variation in order to better match templates of characters. Segmentation is then applied to isolate symbols regions in the input image and recognition step is performed over these sub-images. The basic and most common OCR algorithms for character recognition are based on pattern matching techniques, comparing an image to a stored template on a pixel-by-pixel basis, even though this technique only works with printed characters and has very little tolerance on variations of templates. More recent and flexible techniques [15] rely on extraction of higher level features instead of image samples, such as Histogram Of Gradients (HOG) descriptors, and performing a correlation to decide whether the input represents a certain symbol or not. In the end, the post-processing phase is used to detect words out of character groups and construct the desired output.

2.1.2 Deep Learning perspective

Deep and convolutional neural network models have a long history of application in visual recognition tasks. The first successful applications of

convolutional networks to read zip codes, digits, etc. were developed by Yann LeCun and the best known is the LeNet architecture. Since 2012 the ImageNet Large Scale Visual Recognition Challenge (ILSVRC) [21], which evaluates algorithms for object detection and image classification at large scale, has always seen winning teams that developed advanced neural networks architectures, such as AlexNet, GoogLeNet, VGGNet and ResNet. These architectures are de-facto standards today which proves the huge impact that this technology has in Computer Vision field.

OCR can be seen as an instance of a visual recognition problem and it has been extensively explored in the field of Machine Learning and Deep Learning. MNIST [16] is a very well known database of handwritten digits and it has been widely adopted as a standard benchmark to evaluate performances of new algorithms for digit/character classification. The lowest error rate achieved so far in the task of digit classification from this dataset is 0.23% [17], reaching a near-human performance with a particular architecture that uses a "committee" of 35 deep convolutional neural networks.

2.2 Autoencoder-based OCR

Finding a compact, efficient "coding" for a dictionary of symbols, including a certain degree of variability, it is an interesting aspect that motivated the use of Deep Autoencoders (or Stacked Autoencoders, SAE) for OCR applications as well. Embedded devices and other products for automatic data-capture based on this technology would benefit from such compactness of the coding learned, avoiding the necessity of storing a lots of templates and replacing them only with the classifier weights, which are also more robust to symbols variation.

Once a stack of encoders has been built and trained, its highest level output representation can be used as input to a stand-alone *supervised* learning algorithm: a logistic regression layer can be added on top of the encoders, yielding a deep neural network amenable to supervised learning. The parame-

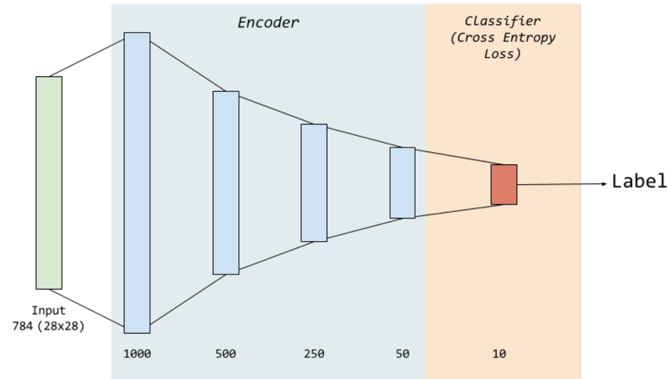

Figure 2.1: Modified architecture of SDAE used to build a MNIST classifier

ters of all layers can then be simultaneously fine-tuned using a gradient-based procedure such as Stochastic Gradient Descent. In this way, we can build a classifier using the unsupervised features encoded in the SAE. Authors in [7] managed to test such classifier over the MNIST dataset using a stack of Denoising Autoencoders, reaching an average error rate on the MNIST test dataset of 1.28%.

We wanted to investigate the same Stacked Denoising Autoencoder (SDAE) architecture in order to train it first on the MNIST dataset and then with a real one. Starting from the SDAE intermediate, lower dimensionality representation of such datasets, we will train a classifier and study its performance in both cases.

2.3 Experimental setup

2.3.1 MNIST classifier

Using the SDAE implementation presented in 1.3.2, we re-arranged it as explained in the previous section in order to train a classifier in a supervised manner, starting from the coding learned during pre-training.

Fig. 2.1 shows the new classifier architecture, obtained by removing the

decoder part of the SDAE and adding a 10 neurons wide layer that will be used as a *Softmax classifier* [22]. This classifier will be trained using the raw output of the highest, innermost level features learned by the SDAE in order to map them to digit labels associated to each sample. This is possible because class score at each neuron in the innermost layer are intrepreted by Softmax function as unnomralized log probabilities of classes and, thus, this layer is trained by minimizing the negative log likelihood of the correct class (using the *cross entropy* loss function) for each input sample.

Training strategies and hyper-parameters impact

In the context of this experiment, we wanted to explore two different strategies used to train the classifier:

- AE pre-training → classifier fine-tuning
- AE fine-tuning → classifier fine-tuning

Note that, for classifying purpose, no fine-tuning has to be performed on the original architecture after pre-training: such strategy would further optimize the weights to lower the *reconstruction error* even more, but in this case it has to be applied only while training the classifier layer, in order to achieve a lower *classification error*, thus a different objective function. Interest in these two strategies is motivated by the fact that pre-training is a pretty slow computation when the number of hidden layer is high, due to the fact that each individual layer requires to be trained independently for several iterations (i.e. training epochs). While the first strategy is the classic training method for such architectures, as explained above, the second strategy is an attempt to train the network as a whole during the reconstruction training phase, thus using a fairly lower number of iterations, and then adjusting its weights for the classification task. Using a fixed amount of iteration while training the Softmax classifier layer, we can then compare the classifier accuracy of one strategy to the other.

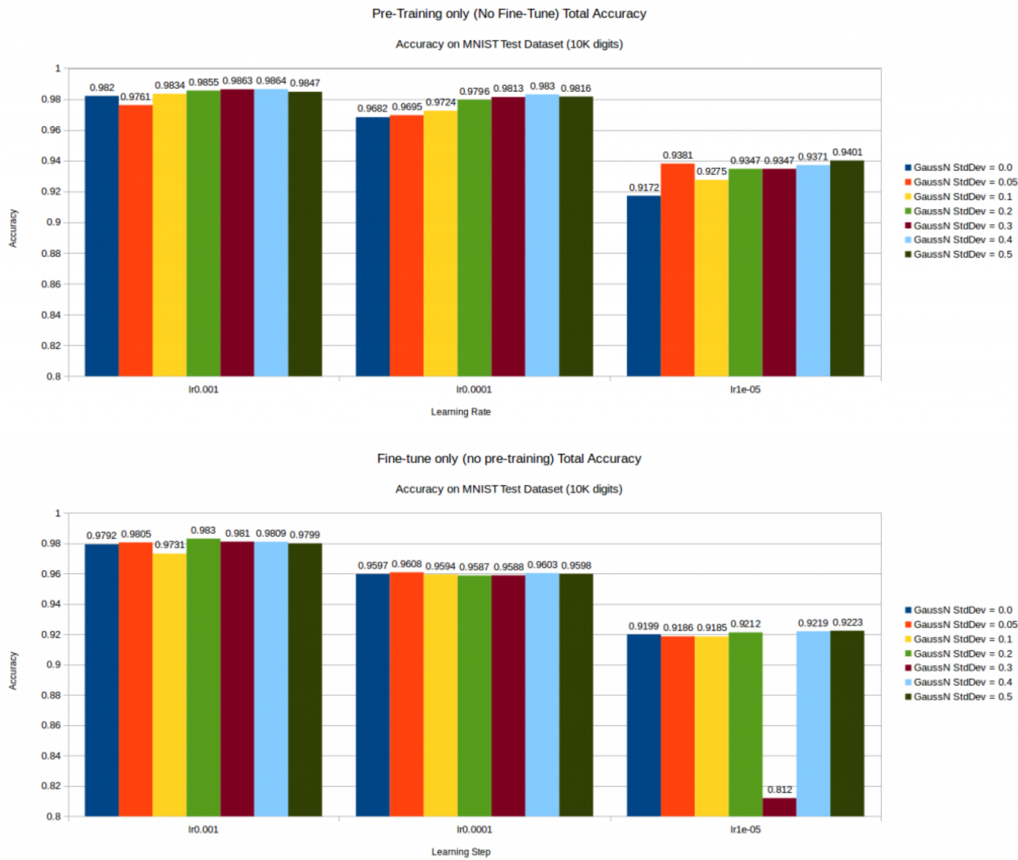

Figure 2.2: Classification accuracy on the MNIST Test dataset using a SDAE-based classifier. Comparison between a classic *pre-training* strategy (top) and *fine-tune* only strategy (bottom).

We used 150 training epochs to train each part of the network for both strategies (i.e. 150 epochs per-layer / whole-network, 150 epochs for the classifier), using a learning rate of 0.001, 0.0001 and 0.00001 for the SDAE training and an additional factor of 0.1 applied to the learning rate when training the classifier layer. Moverover, an increasing level of Gaussian noise has been applied, to see if it really improves the classification performances, letting the network generalize better while learning the coding.

Fig. 2.2 shows the outcomes of such strategies for each combination of hyper-parameters value. From the strategy point of view, the classic training

approach leads to a higher accuracy in almost every case, yet the fine-tune only strategy is still comparable in terms of accuracy. This let us imagine a plausible application of such strategy in contexts where a relatively small percentage of accuracy can be sacrificed in favor of a much faster training.

For what it concerns the learning rate, higher learning rates converges earlier to a good accuracy as expected, but lower ones should let us reach a better local minima on a long run (recall that the number of training epochs here is fixed). A good strategy in this case, probably already implemented in TensorFlow optimization routines, could be using a large learning rate at first and then lowering it down gradually while approaching a minimum region while optimizing weights.

In the end, from the noise level point of view, in the classic training approach it effectively leads to slightly better classification accuracy as the noise increase: when the amount of noise is set to $\sigma = 0.3$ and $\sigma = 0.4$ it seems to effectively increase the accuracy rate with respect to other cases.

The best performance in terms of accuracy reached here has an error of 1.36% on the 10.000 samples MNIST test set, which is very close to the results reported in [7] with the same architecture.

2.3.2 A real-world case

After training the MNIST classifier, we had a nearly complete TensorFlow implementation of a general classifier based on a Stacked Denoising Autoencoder. The next step for my work at Datalogic was testing this architecture on one of their clients' dataset, in order to compare what is the classification performance of this neural network with respect to their best performing product, the computer vision based library HOG-OCR [15].

Dataset description

The dataset chosen, whose company name cannot be mentioned here due to NDA, is one of the hardest datasets among Datalogic's clients: it is composed by 640x480 pixels gray-scale pictures of snacks boxes with the

Figure 2.3: Abstract model of pictures in the dataset.

expiration date and the lot code printed on top of it. Box are marked with such informations using a dot-matrix printer in a rectangle shaped white area (see Fig. 2.3 for reference). The complete dataset is composed by several folders, each one containing pictures taken from the same day of production, for a total amount of 39078 pictures. The images were originally unlabeled.

This client needs a fast and reliable solution in order to correctly read out such informations from pictures with an accuracy of 99.9%. What is most challenging about this dataset is that printed characters present a high degree of variation over time:

- they appear stretched in and out in both horizontal and vertical axis;
- they present a certain degree of rotation clockwise and counter-clockwise;
- text position, with respect to the intended area on the box and on the picture itself, continuously changes over time.

HOG-OCR performance

Currently, HOG-OCR uses a sliding window approach to extract a correlation map of the input image. The correlation has to be done using every character template, including their variation, for each location in the image.

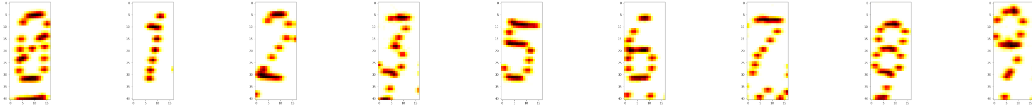

Figure 2.4: Digit samples from the synthetic dataset.

This means it has to store lots of templates to work properly on datasets with high variability like this one. HOG-OCR best performance in *recognition mode* on the complete dataset is around 57% (at image level) using 37 templates from the first 5 folders.

The amount of variability makes it really hard for HOG-OCR to process the dataset, so the goal is to replace the HOG-OCR underlying classifier with a more robust and efficient one.

Using a synthetic dataset

A common requirement in Datalogic's applications is a limited time to train the classifier: this means that, in high variability conditions, it is possible that not all the variations of samples are covered by those collected. In general, training any type of Deep Learning model requires a lot of data. In Datalogic's applications, they typically do not have enough data available (mainly because they cannot expect their customers to label an very large amount of training data.

Datalogic overcomes the limited amount of training data by augmenting the data with *synthetic* samples, in order to obtain a dataset with a similar value distribution as the ones in the real, unseen data. For this particular dataset, each sample is labeled as one of the following classes: `background, dot, 0, 1, 2, 3, 5, 6, 7, 8, 9`. Starting from the same character samples used to train HOG-OCR on the same problem, they've generated a *synthetic dataset* for an amount of 17590 synthetic samples, by applying transformations to the original samples (rotation, scaling, affine or perspective transformations, etc.). The key to obtain good performances from this approach is to have a training dataset that covers all possible variation of the samples in the real dataset. An example for each digit in the synthetic

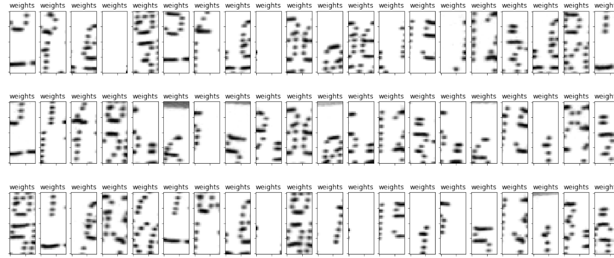

Figure 2.5: Background class samples.

dataset is presented in Fig. 2.4.

The `background` label in the synthetic dataset needs further attention: it contains samples generated to be either all white patches or presenting partially occluded digits. The idea when generating this class was to capture patches in the real images with uniform coloration and the concept of "spaces" between digits, which is known to generate lots of False Positives. Fig. 2.5 shows some background samples extracted from the dataset.

It is important to note that this background class doesn't generalize very well for other background elements in the client's dataset (e.g. barcodes, qr codes, box borders, other printed elements or drawings on the box), as we will see later, but it will be used anyway for a first evaluation of our solution. Moreover, in the context of a real Datalogic application, background typically changes very often and, even if those elements were considered while building the synthetic dataset, some general way to handle unseen input elements (typically resulting in a False Positive detection) has to be considered.

Training

The synthetic dataset is splitted into 3 parts: training (10554 samples), validation (3518 samples) and test (3518 samples). Training dataset is used to train the neural network, while validation and test sets are generated by samples that are not present in the training dataset. Validation and training sets are used as a stopping criterion while training the classifier layer: when it reaches a desired accuracy on the validation set (99.9%) and on training

set (99.99%), the classifier training is stopped and the final accuracy on test dataset is then computed. Each training epoch consists in a complete processing of the training dataset, divided in mini-batches (i.e. groups of 128 images) randomly ordered at each epoch.

Dropout regularization

Moreover, during classifier training, *dropout* [19] regularization has been applied with a 50% of probability to *shut down* neurons at each inner encoding, in order to reduce overfitting by training only a partition of the network at a time, thus making the network temporary "smaller". As the authors says, "it provides a way of approximately combining exponentially many different neural network architectures efficiently", similarly to the impractical technique of averaging multiple models ("ensemble"), which shows better performance in most machine learning tasks (e.g. ensemble training is the intuition behind random forests or gradient boosting decision trees). Training a neural network with dropout can be seen as training a collection of 2^n thinned networks with parameters sharing, where each thinned network gets trained very rarely, or not at all.

2.3.3 Hyper-parameters optimization

According to results obtained in 2.3.1, we kept a similar network architecture to solve a character classifying problem on the synthetic dataset. A hyper-parameters optimization has been performed, adjusting network size and learning rate so that the desired accuracy is met and classifier training converges in a reasonable time for the reference dataset.

We obtained a configuration able to reach 99.9% of accuracy on the validation set in around 30 classifier's training iterations. Our best performing classifier embodiment for this dataset was the following, at this point:

- **Architecture**: 4-layer SDAE (800, 500, 300, 100 neurons)
- **Classifier**: *Softmax* multinomial logistic regression layer (11-classes)

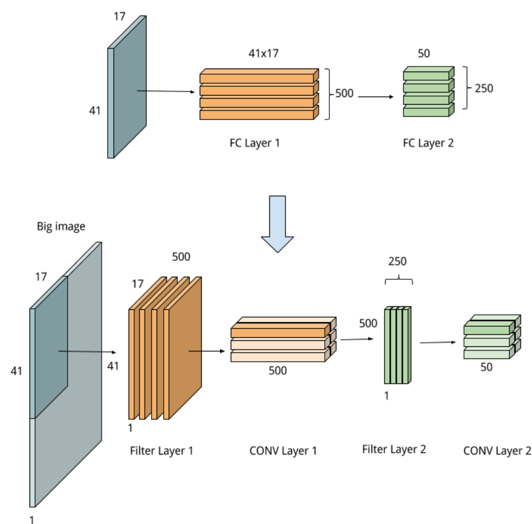

Figure 2.6: Graphical explanation of Fully Connected (FC) to Convolutional (CONV) layer conversion for a 2-layer network.

- **Training epochs**: 150 (fixed during pre-training, max 3000 during classifier training)

- **Learning rate**: 0.0001 (SDAE pre-training), 0.00001 (Classifier fine-tuning)

2.4 Testing on real images

Now that we have trained our classifier on the client's dataset, let's apply it on the whole images.

Since there's no ROI to search within for characters, we need to process the entire image at each possible sample location, classifying that input position. Following this approach, we should have computed a full forward step of SDAE-C at each position of the original image.

2.4.1 From fully connected to convolutional layers

In order to test this classifier and recognize digits on real (bigger) images in an efficient way, I managed to convert the encoding forward function of the SDAE from fully connected to convolutional.

Each neuron of any hidden layer can be seen as a convolutional filter [9]. The forward pass from one layer to another gives exactly the same output using both fully connected and convolutional way to compute it, but this conversion allows us to "slide" the entire classifier network very efficiently across many spatial positions in a larger image, in a single forward pass. A graphical explanation is given in 2.6. Using this approach with a trained SDAE-C, the whole convolutional forward pass lead us from the original 480x640x1 input image to a 440x624x11 volume output, which is the prediction on each of the 11 labels for every single input crop evaluated by the convolutional forward pass. This volume is given by the convolution applied on the bigger image, whose size can be predicted at each convolution layer in this way:

$$\begin{gathered} input : W_1 \times H_1 \times D_1 \\ output : W_2 \times H_2 \times D_2 \\ W_2 = (W_1 - F_w)/S + 1 \\ H_2 = (H_1 - F_h)/S + 1 \\ D_2 = K \end{gathered}$$

where $W_x \times H_x \times D_x$ is the input/output volume (width, height, depth), K is the num. of filters, S is the convolution stride, F_h and F_w are, respectively, filter height and width.

2.4.2 Classifier evaluation

After computing the convolutional forward step of the entire image, which results in a mapping of label scores assigned by the classifier at each input

Data: $M = \langle x, y, h \rangle$ where x, y are position in the input image and h are the labels score at that position;
Result: List of classifier activations;
$detections = \{\}$;
foreach x, y **do**
    foreach h **do**
        $j = \text{argmax}(M[x, y, h])$;
        if $M[x, y, h]_j \geq$ *threshold* ***and*** $\langle x, y \rangle$ *is a local maxima* **then**
            insert $\{\langle x, y \rangle = j\}$ in *detections*;
        end
    end
end
return *detections*;

Algorithm 1: Extract a list of detections from classifier output. (Note: h is a vector of 11 class scores at each location)

image location, Algorithm 1 is used to extract a list of detections from the input image, based on the classifier score values.

Accuracy

The number of True Positives (TPs), False Negatives (FNs) and False Positives (FPs) is obtained comparing the classifier output with the groundtruth information on each image. Specifically, for each detection we compare the area of the sample found by the classifier with the one from the groundtruth label. If the detection overlaps with the groundtruth label for at least 50% of their areas, it is then considered a TP. Algorithm 2 describes this procedure and it is used to count the number of TP, FP and FN in the image.

The goal here is to obtain a configuration of the network able to get 99.9% of digit recognition rate in real images and as few FP as possible, since they could be filtered at a successive stage using some spacing model and knowledge-based search.

Data: *detections*: list of detections from the classifier in the form of $\langle x, y \rangle = j$; *groundtruth* labels on the input image;
Result: Number of FP, FN, TP

```
fp, fn, tp := 0;
foreach label j do
    foreach groundtruth location with label j do
        count := 0;
        foreach d in detections do
            if d is labeled as j and d have an overlapping patch area of
              at least 50% with the groundtruth one then
                count++;
            end
        end
        if count > 0 then
            tp++;
        else
            fn++;
        end
        fp := number of detection with label j that aren't TP;
    end
end
return fp, fn, tp;
```

Algorithm 2: Compute the number of TP, FP and FN in an image based on classifier output and groundtruth info.

2.5 Experimental setup

In order to test the classifier, we've used a set of 16 labeled images from the client's original dataset, chosen among separate folders to embrace most of the characters and background variations. This subset of images can be considered a good test candidate under the observation that each folder presents very similar features in terms of character distortion and background variation.

The number of labeled samples on each image varies between 8 and 9 digits belonging to one of these classes: `0`, `1`, `2`, `3`, `5`, `6`, `7`, `8`, `9`. When a sample is correctly recognized, we will count it as TP. The rest of the images locations (i.e. crops) should be classified as one of the two background classes, that is `background` or `dot`[1]. For any input location classified as a digit that doesn't match with a labeled sample in the image, it will be counted as a FP detection. Since no ROI is defined, the whole image will be processed by the classifier at each input location.

In the next section we will first evaluate how different noise levels affects classifier performance, by means of digit recognition rate (i.e. how many labeled samples have been correctly classified) and number of FP detection for the whole subset of images. Then, we will propose some solutions to improve its performances.

2.5.1 Basic SDAE-C performance

Noise impact

The following table shows how noise at different scales affect the classifier performance on the real dataset, in terms of number of digit recognition rate and number of FP per image, as explained above. Results are intended for the subset of 16 test images extracted from the real dataset previously introduced. For each configuration, at different noise levels, results are averaged over 5 separate classifiers, trained from scratch using 150 epochs of

[1] `dot` class consists of dots samples that can be found between digits in test images.

Figure 2.7: Comparison between features learned at first layer of the SDAE-C using $\sigma = 0.2$ (left) and $\sigma = 0.4$ (right). Using more noise during training let the feature be more general but more noisy as well, which makes them fire the neurons even with fewer constraints on the input features, but also increases the chance of wrong activations.

pre-training at each SDAE layer:

Noise (σ)	Avg. Digit recognition rate	Avg. FN (142 total)	Avg. FP per image
0.0	97.888%	3	98.86
0.1	97.888%	3	237.06
0.2	98.308%	2.4	515.90
0.3	98.591%	2	1016.85
0.4	98.591%	2	1339.36
0.5	98.591%	2	1534.51

We can see that, on the average, increasing noise tends to improve the accuracy in terms of a higher digit recognition rate on these images, but the number of FP per image increases at a much faster rate. Observing activations from each test image, FPs are due to two main factors: (1) natural input similarities with features learned by the autoencoder and (2) more noisy templates learned by the SDAE, which have the bad side of generalizing also fuzzy/noisy patterns to wrong classes (see Fig. 2.7 for a close-up comparison of features learned at first layer neurons with low and high noise). On

the other hand, FNs are mostly due to particular character distortion that haven't been taken in account in synthetic samples.

The best configuration in this case was obtained using $\sigma = 0.2$, resulting in 140/142 correct detection. When applying no noise on training samples, none of the classifier configurations obtained was able to get more than 139/142 correct detection, which again demonstrates the noise role in improving generalization over unseen variation of samples. Fig. 2.8 shows what features neurons at first layer have learned in one of these runs. It's pretty evident that, at the first hidden layer, the network is learning several templates for samples in the dataset, as already observed for the MNIST case.

Effect of `background` class

Fig. 2.9 shows activation maps for each label on one of the 16 client images, where all characters have been correctly detected (FN = 0). Looking at the figures, we can see that activations for class `0` and `5` happens more often than others on picture elements as barcodes, qr-code and other printed elements. This means that, when the classifier is processing those elements, it better approximates them as `0` or `5` samples, instead of labeling them as `background`. This happens because, as explained before, there are no samples of such elements in the dataset used to train the network and they are therefore associated to the class that better fits them, according to what neural network has learned. Besides, `background` class seems to approximate pretty well the model expressed for non-character images in this application and can be therefore used as a starting point to build a more robust dataset.

2.5.2 Increasing SDAE-C performance

New stopping criterion

One way to help the network generalize better on character inputs is to include the test set accuracy, together with the validation one, as a stopping

Figure 2.8: Weights visualization of 200 neurons at the first layer of the SDAE-C (trained using noise level $\sigma = 0.2$).

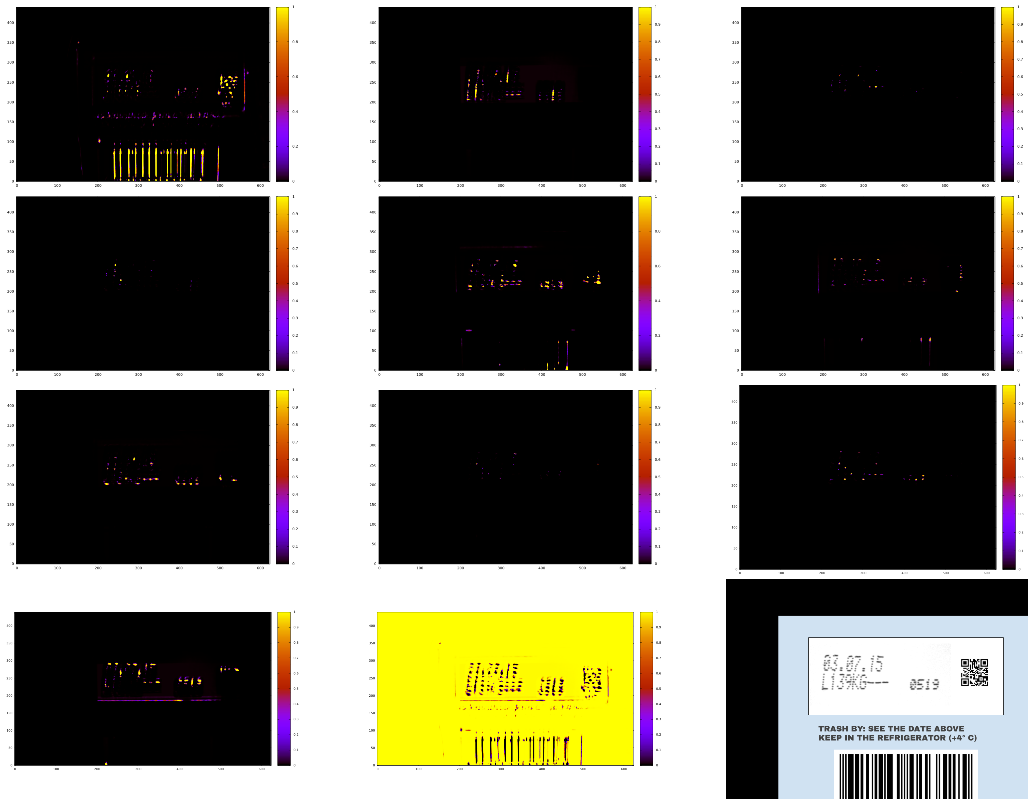

Figure 2.9: Activation maps for each label obtained with a SDAE-C trained at noise level $\sigma = 0.4$. The images are referred to labels in this order (left to right, top to bottom): `0`, `1`, `2`, `3`, `5`, `6`, `7`, `8`, `9`, `dot`, `background`. Last figure is left only as a reference for activations positions.

criterion when training the classifier. Precisely, training has been stopped when 99.9% of accuracy was reached in both datasets, instead of using validation set alone. Before applying the new stopping criterion, observed average accuracy reached on the test dataset at the end of training was always between 99.7% and 99.8% for all runs. This approach has the benefit of fine-tuning weights in order to reach a configuration that is more robust on a wider range of samples, but it also needs more training epochs to meet this requirement: this is commonly known to cause overfitting, that is keep updating weights without learning anything meaningful and reducing the classifier overall accuracy (i.e. it generates more FPs).

More pre-training epochs

Also, more pre-training at each layer means better coding of the full dictionary of samples in the training dataset. Keep improving the reconstruction ability of the Autoencoder let it learn stronger features and confuse less fuzzy patterns with characters, reducing the amount of FPs.

Role of dropout

As a final note, dropout regularization alone helps reducing FPs with a factor of around 2 or 3.

2.5.3 SDAE-C final considerations

We performed further experiments using more training epochs (250 instead of 150) and the new stopping criterion explained above, reaching several different configurations of the network able to recognize 100% of the characters in the chosen subset of 16 images using the given dataset, most of them obtained when training the Stacked Autoencoder with a denoising approach. However, the weak point of this classifier solution is the high rate of False Positives as the noise increase. In this kind of OCR application, high variability of input is known to be cause of FP detections and, even if they can

be filtered using some spacing model as in the strategies used in HOG-OCR library [15], hundreds of FPs like in case of this dataset would made a post-processing phase of such detections very expensive, due to the high amount of data that need to be processed.

Since elimination of False Positive is a very common issue in Datalogic's tools (and in HOG-OCR as well) our work started focusing more on this kind of application for Autencoders. We will present now a solution to decrease the number of False Positive at a classifier output, first by applying it on the best embodiment of our SDAE-C and, in the next chapter, proposing it as a complete stand-alone solution.

2.5.4 Additional Autoencoder to deal with FPs

One of the major contribution of this thesis is the idea of using the original Autoencoder used to train the classifier as a *verifier* to get rid of False Positives. This approach is based on the observation that an Autoencoder is trained to reconstruct at its best something that he has *seen* while training, while reconstructing the best approximation of what it knows if it's something it has *never seen.* The idea is the following:

1. Save the weights of the original SDAE (right before the classifier layer training starts);
2. After using the classifier on a real image and the detection list is produced, reload the weights of the original SDAE;
3. For each detection in the list, make the original SDAE reconstruct the locations in the real image corresponding to each detection;
4. By looking at a reconstruction, it's possible to tell whether it is a *good* detection (i.e. reconstruction cost is *low*) or a *bad* one (i.e. reconstruction cost is *high*)

The point of using the original Autoencoder implementation as a second stage classifier is novel to us and particularly interesting because it could also

be used as plug-in for any other classifying pipelines, beside the one proposed in this work.

2.5.5 Improved SDAE-C performance

We performed again the experiment in 2.5.1, this time applying AEVs to the SDAE-C classification output. In this case, we simply found empirically a threshold value based on the average reconstruction error (observed while training the SDAE) large enough so that all the TPs are kept for all the character labels.

The following table shows the classifier performance after using this approach. For each configuration, at different noise levels, results are averaged over 10 separate classifiers, trained from scratch using improved training strategies explained in 2.5.2 in order to reach a better digit recognition rate:

Noise (σ)	Avg. Digit recognition rate	Avg. FN (142 total)	Avg. FP per image
0.0	98.38%	2.3	105.58
0.1	99.22%	1.1	117.96
0.2	98.87%	1.6	134.33
0.3	99.50%	0.7	135.33
0.4	99.64%	0.5	152.18
0.5	99.29%	1	157.84

By applying an Autoencoder Verifier (AEV) to the output list of detections, we were able to reduce dramatically the number of FPs generally by a factor of 3, effectively removing most of the background false activations (e.g. almost 0 FPs on barcodes and other non-characters samples for all the 16 images). From these results we can also see also the benefits of the new stopping criterion and increased amount of training epochs at pre-training, obtaining more configurations able to reach 100% of digit recognition rate on the 16-image dataset.

Still, noise improves generalization at character level, but we have to keep in mind that FP rate is really high in this case without AEVs application,

due to both noise effect on templates and overfitting introduced with the new stopping criterion for classifier level training.

Defining when the reconstruction can be considered *good* or *bad* is not an easy task. This simple solution is found to be effective on the task of reducing FPs at a classifier output, but not at removing all them: the fact that there are still a hundred of FPs means that those ones have a reconstruction cost below the threshold and are therefore considered good detections.

From a single AEV to class-specific ones

By measuring the average reconstruction error over dataset's sample with the Autoencoder Verifier, we noticed also that each label had a different average reconstruction error value, which means that the SDAE doesn't learn to reconstruct all the samples with the same level of accuracy. This is expected, since a single network is trained to learn the coding for several different classes and thus there's a high chance to be imbalanced on the amount of features learned per class of character.

This suggested the idea of training several different AEV for each label instead of a single one, having in mind the goal of obtaining a smaller network highly specialized in learning a code for a specific class of sample, leading to a smaller reconstruction error and, thus, a higher accuracy in defining a threshold value that can be later used to tell if the input to the AEV is effectively something it has learned or not.

At this purpose, a set of single layer AEVs have been trained to be tested on the SDAE-C architecture.

2.6 Performance on complete folders

Using our best configuration obtained for the 16-image test set (100% recognition rate and the lowest average of FPs per image), we wanted to test the accuracy of our classifier on some of the complete folders of the client's real dataset, in order to see what's SDAE-C performance on a bigger

number of images and in presence of variation of data along the same day of production.

Labeling real images and classifier accuracy

Since the client's dataset was not already fully labeled at the time these experiments were performed, we dumped HOG-OCR detections from every correctly labeled image to extract the groundtruth information for the images in real dataset's folders. We then applied SDAE-C classification over the complete images and used groundtruth to test the SDAE-C accuracy at character level on this wider set of real images. For 7 folders, the classifier reached 100% of character recognition except one where it scored 99.95% (4 FNs in the whole folder). For all other folders, the accuracy was lower: typically this was due to same characters being missed over and over again throughout the images in the same folder, which means that those particular distortions of the sample were not covered in the synthetic dataset.

Without any detection filtering, the average number of FPs per image with this classifier configuration was in the order of one thousand. We will now see how AEVs could help reducing such FPs using different cutoff levels.

AEVs application

Using the synthetic dataset, we trained a single layer (shallow) AEV for each digit class and extracted the average reconstruction error at the end of training. The average itself can't be considered a good threshold, since there will be samples uncovered in the synthetic dataset that will result in a higher reconstruction cost than the ones observed while training. Therefore, we started looking at how the recognition rate changed while adding a +0.5 tolerance (ranging from 0.0 to 9.5) at each run to every AEV's base threshold value, in order to compare the TP rate and the average number of FP per image during each experiment, as the tolerance on reconstruction error increases. The goal is to find a common level of tolerance to be applied to each base value threshold to keep all the TPs and reject more FPs as possi-

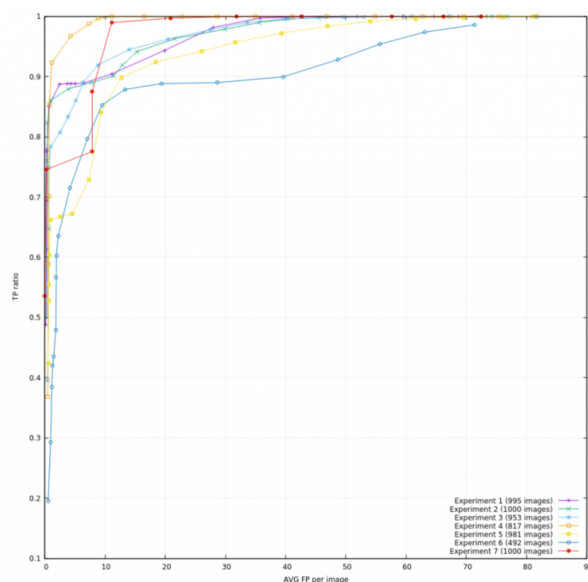

Figure 2.10: ROC curve to show TPs rate over average FPs per image at dataset's folder level. Starting from first, each successive point on the curve indicates an increment of 0.5 on the threshold used to discard FPs.

ble. Each experiment refers to the SDAE classification + AEV verification outcome performed over a folder of the client's dataset (we've chosen to test only the 7 best performing folders).

Fig. 2.10 shows the ROC[2] curve obtained in such folders, showing the True Positive rate over number of average False Positives obtained per image, as the level of tolerance applied to the threshold increases. As we said, without any filtering the SDAE classifier in this configuration produced an average of around 1000 FP per image: after using AEVs, we're able to get the maximum TP rate on almost every experiment with a tolerance of 9.5 and to reduce to less than 90 average FP per image in the worst case scenario (experiment 5) and to 12 in the best one (experiment 4).

As expected, AEVs work better when the samples in the image are well covered in the training dataset (e.g. experiment 4), using a relatively low tolerance on the digits mean reconstruction error, while they need a higher

[2]Receiver Operating Characteristic

tolerance (throwing less FPs away) for other folders.

2.7 Conclusions

Even if using SDAE-C has been shown to reach a satisfying character recognition rate in conditions of high variability of samples to be recognized, the high number of False Positives make it really hard to be used in practice in a real Datalogic's applications. In this case, improving the synthetic dataset generation is the primary key to get the classifier generalize better and achieve better performances. Nonetheless, in conditions of lower variability of samples, SDAE-C is still suitable to be used as a robust classifier.

The AEV approach showed some promising results in the task of reducing the number of FPs at the classifier output. An improved general strategy based on AEVs is discussed and tested in the next chapter.

Notes on timing

The time needed to train the SDAE-C on the synthetic dataset is around 7.5 minutes (6.6 minutes to pre-train all layers and 57 second to reach the desired accuracy while training the classifier) using TensorFlow with GPU acceleration on a machine equipped with Intel Xeon (R) CPU E5-1620 v3 3.50 Ghz x 8 and NVidia GTX 1080 GPU.

A convolutional forward pass performed to process an 640x480 image takes around 0.7028 s with such configuration. The latter timing, in particular, doesn't make SDAE-C suitable to be applied in a real-time application, which highlights the needing of a more efficient implementation or some technique to avoid processing the entire image (e.g. some ROI identification strategy).

Chapter 3

Autoencoders as second level classifier

The high number of FPs motivated also the idea of using the SDAE architecture as a second stage verifier and add it to the classifier output, in order to reduce (or eliminate) false activations and improve the classifier output.

The key aspect of such approach is Autoencoders ability to reconstruct the things they "have seen" during training: if we input the AEs with something it has never seen, it will try to reconstruct something he knows, giving a higher reconstruction error. A meaningful way to measure this reconstruction error is the RMS error per pixel intensities: the lower the error, the higher its reconstruction accuracy.

3.1 Training a SDAE for each class

Using a smaller network, we can use several SDAEs, here called Autoencoder Verifiers (AEV), to verify the classification output of any classifier, if we know classes and have access to a reasonable number of samples of any given class.

We perform the following steps:

1. Learn a separate AEV for each class (characters, for instance);
2. Verify that a detection of, say, an “A”, is really an “A” by inputting the crop of the character to the AEV and compare the output to the input, measuring its reconstruction error.

The training procedure is the usual one explained so far to train a SDAEs to reconstruct its input, with the only difference that we train multiple, smaller SDAEs for each class instead of training a single big network for the complete dataset. The reason of using smaller networks is twofold: (1) each AEV will learn a single class of the complete dataset, so the number of features that has to be coded is way lower than the classifier case, and (2) we have to apply such AEVs several times for a single real image in a limited amount of time, in order to verify all the detections on it - a smaller network means fewer operations needed to compute the encoding/decoding forward pass.

A layer-by-layer pre-training is performed using samples from a single class, applying a small scale of noise ($\sigma = 0.1$) to enhance features learned by each AEV, followed by fine-tuning of the entire network to further optimize the weights. At this point, each AEV is fully trained and it is supposed to reconstruct accurately (i.e. with a low RMS error) each input that has a similar distribution of values along its features as the ones it has “seen” while training.

3.2 Define the reconstruction threshold

As already explained in the previous chapter, typical applications have to deal with high variability of samples. All possible distortions, variations and corruptions of samples usually could not be taken in account even if we augment the original dataset, due to the limited amount of training data. This means that not only wrong samples, but also samples belonging to a specific class unseen while training that particular AEV will result in a higher reconstruction error, with respect to the average one observed during training.

Thus, we need a way to estimate a threshold, based on the reconstruction error observed while training, that would likely keep all the correct samples (even the ones unseen while training) and possibly reject all the others or most of them. One could simply think about the average RMS of the last training iteration as a possible threshold, but this isn't an accurate choice: while a RMS error below the average it's a clear sign of a correct classification, a sample that results in an error above the average wouldn't necessarily be a wrong one.

3.2.1 Modeling reconstruction error with a Normal distribution

We could model the reconstruction error observed while training each AEV with a normal distribution, centered on the average RMS error of the last training epoch: reconstruction error of data observed while training will fall in a given range around the average that can be easily estimated while training by computing the standard deviation σ along with the average RMS error for the observed dataset. This let us set a higher threshold for AEV_j (Autoencoder Verifier for class j) using the upper limit of the computed distribution:

$$t_j = \mu_j + 3\sigma_j$$

This let us also take in account correct samples that result in a reconstruction error above the average, but still in the range of observed data.

Still, this is not a really useful threshold because of the issue explained before: a correct, unobserved sample could still be rejected if its reconstruction is higher, as it is likely to be since AEV is heavily tuned on training samples.

3.2.2 Verification dataset to define threshold

In order to define a meaningful threshold for the verification step, we can use *unobserved* data from a verification dataset. We simply apply our trained

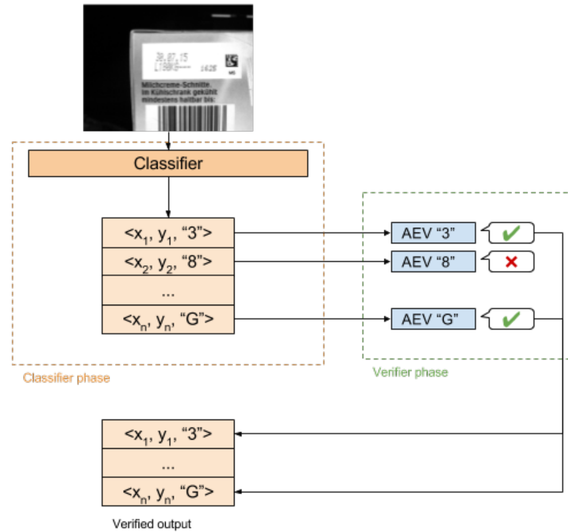

Figure 3.1: An example of OCR pipeline with the additional verification step, performed by Autoencoder Verifiers on the classifier output.

AEVs on such dataset and compute a new μ_j and σ_j for each class j, which will form a distribution having a reconstruction error range that will better reflect data observed in the real application.

Nevertheless, even if this approach gives us a good approximation of the RMS error threshold to be used to discern correctly labeled samples from, it will be based only on variation of the data in the training and validation dataset. The only way to ensure that these thresholds are good enough to represents the distribution relative to the reconstruction errors and obtain good performances is to use a training and verification dataset that covers most of the possible variations of samples in the real dataset.

3.3 Using AEVs to verify classifier outputs

Now, we have a set of trained AEVs for each class of samples, with an associated threshold value that can be used to distinguish between samples that doesn’t belong to a certain class, based on their reconstruction error.

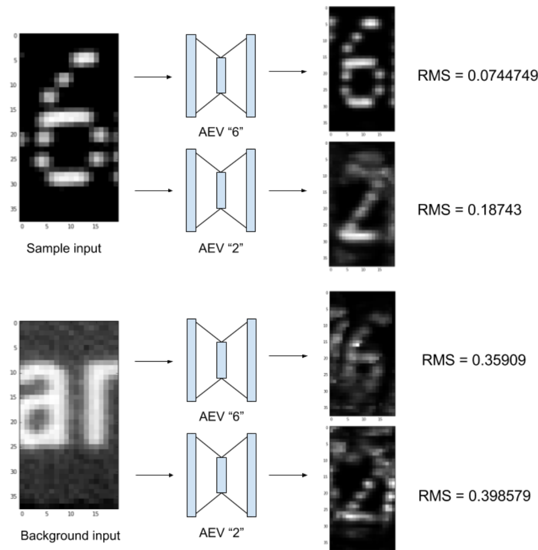

Figure 3.2: Difference in RMS error between a correctly labeled sample and one that we would like to reject. The different outcomes using two AEVs are shown for each sample.

Since these objects can are trained separately from the classifier (and it would be even better to use two different synthetic dataset), AEVs can be plugged-in to a generic OCR system pipeline to prune/verify a character detection of the classifier output. More specifically, given a list of n detections, in the form of triplets $\langle x, y, h \rangle$ where x, y are coordinates in the input image and h is the label assigned by the classifier, we verify each labeled detection by re-submitting the input to the Autoencoder Verifier trained for that class (see Fig. 3.1).

The approach is as follows:

1. Given coordinates in the processed input image of a labeled character detection to be verified, crop an area around the activation area in the original input image;

2. Use the cropped image as an input to the trained AEV for that specific label j;

3. Compute the RMS error between reconstructed output $\mathbf{z}$ and input $\mathbf{x}$ in this way:

$$e_{rms} = \sqrt{\frac{1}{N}\sum_{i,j}(z_{ij} - x_{ij})^2}$$

4. If $e_{rms} \leq t_j$, accept the activation as correctly labeled, otherwise discard it as a FP.

This approach is based on the observation that if the input to the AEV wasn't really, say, an "A", the AEV_A would not be able to reconstruct the input faithfully - it will reconstruct the best approximation of what the input should be if it was supposed to be an "A", thus giving a quite large RMS error. This error should be even higher than the one computed out of unobserved, correct samples from the validation dataset used for AEV_A. Fig. 3.2 shows some output examples when using this approach on correctly labeled samples or wrong ones.

3.4 Experimental setup

A set of smaller networks has been trained using a newly generated synthetic dataset. At this time, the dataset is divided in two parts: the training dataset and the verification one, that will be used to compute the reconstruction error thresholds for each class as explained above.

For all AEV configurations tested, layer-by-layer pre-training and fine-tuning phases are carried on for 150 epochs each, applying Gaussian noise with scale $\sigma = 0.1$. The learning rate is set to 0.001.

3.4.1 Visualize reconstruction errors on a real image

As a first experiment, we wanted to see if the application of a single AEV, trained only on a specific class, effectively leads to a smaller reconstruction error at desired samples location when applied to an input image.

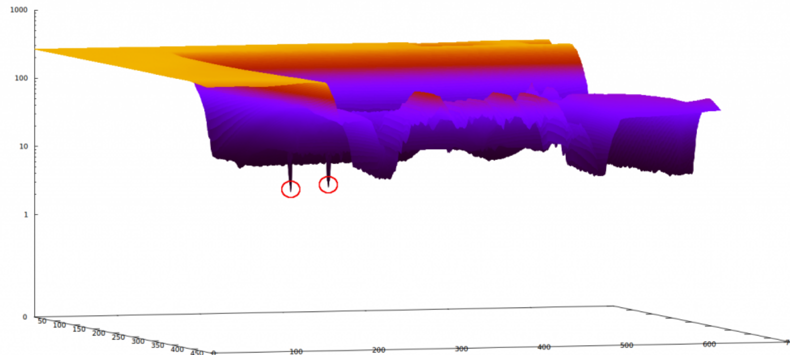

Figure 3.3: Visualization of the reconstruction error obtained using a AEV trained on class "0" at each location in a real input image.

Fig. 3.3 shows the reconstruction error on the vertical z-axis (using a logarithmic scale) obtained by applying a AEV trained on the label "0" on a real image from the client's dataset. The horizontal xy-plane corresponds to the pixel locations in such image at which the reconstruction process has been applied. The lowest reconstruction error peaks are highlighted by the red circles: those are exactly the locations in the original image where the sample 0 is found.

3.4.2 Shallow vs. deep architecture

Using this visualization technique, we proceeded to test if a shallow architecture (i.e. with a single hidden layer) would achieve the same performances of a deep one, which is supposed to learn a more robust and hierarchical code for the input dataset. Specifically two sets of AEVs have been trained:

- **Shallow**: 1-layer DAE with a 100 neuron hidden layer;
- **Deep**: 3-layer SDAEs with hidden layer sizes of 100, 50 and 25 neurons;

In fact, it has been observed empirically that the deep AEVs lead to a better performance in terms of lower reconstruction error (during both

training and tests) on TP locations of real images and, in some cases, higher reconstruction costs on all other locations. This is exactly the behavior we need in order to make this approach work the best, clearly separating the good samples from the misread ones.

3.4.3 Practical application of AEVs

In order to get a low reconstruction error for TPs, we need the sample in the cropped area to be in a "similar" position with respect to the ones used to train a specific AEV. If we consider the x, y[1] location of a detection produced by the classifier, when cropping the area it could happen that the patch produced has the sample in a different position (misplaced or partially occluded) with respect to the samples in the training set. Moreover, other characters or background parts could appear on the patch extracted: any type of variation in terms pixel values with respect to the samples observed during training will alter the RMS error, raising its value to the point that even correct detections can be discarded.

Since we could not quantify the detection displacement compared to the size and features of samples in our training datasets, we perform the AEV reconstruction on more than a single crop, each of them being generated from positions surrounding the original detection coordinates of the input image, essentially processing a bigger crop. The surrounding area is determined by adding a fraction of the training samples' height and width around the original detection coordinates. Each pixel in the enlarged area will result in an additional patch to evaluate, as if we were "sliding" the AEV all over this area. Specifically, given a fraction $0 < f < 1$ (e.g. $f = 0.5$) and training samples width w and height h, we can easily determine the number of crops as:

$$C = \lfloor fhw \rfloor$$

[1]Typically, this location corresponds to the top-left pixel of the detected sample, but it could also be in other parts (e.g. located in the center of the activation).

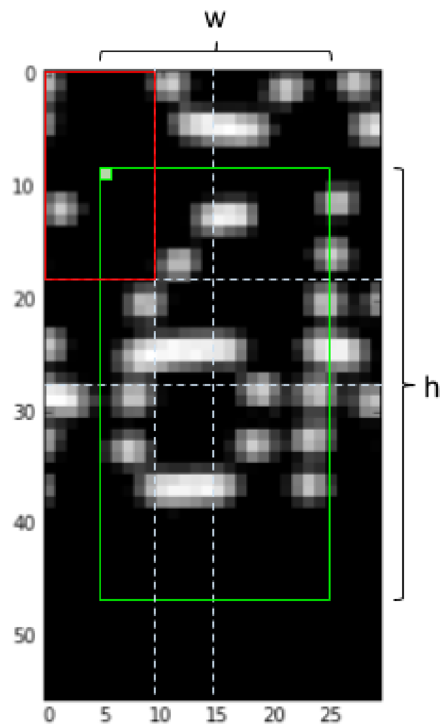

Figure 3.4: The green rectangle is the original crop, generated from the detection coordinates. The red rectangle is the area around the original detection, given $f = 0.5$. Each pixel in this area will result in a new crop to evaluate. The whole image is the total area extracted from the input that will be processed.

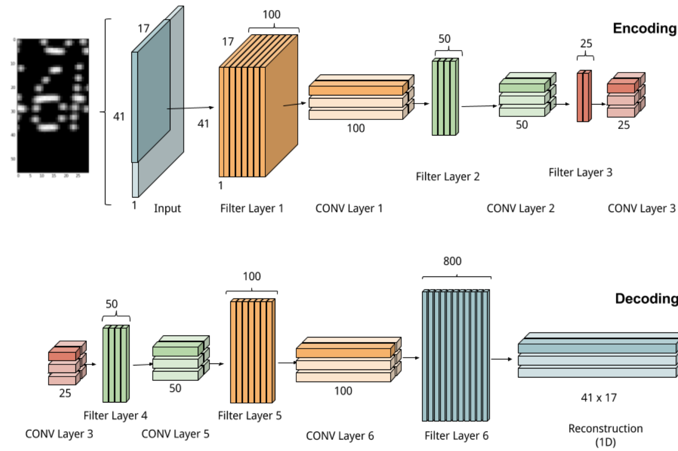

Figure 3.5: Encoding (above) and decoding (bottom) steps performed over the wider crop area using a convolutional approach.

Fig. 3.4 visually explains how the crops are determined. After reconstructing and evaluating RMS on each of these crops, only the minimum reconstruction error is kept and compared to the threshold to accordingly reject a detection or not.

3.4.4 From multiple fully-connected reconstructions to a single 2D convolution

In order to improve performances of these evaluations over the extracted patch area, an approach similar to the one explained in 2.4.1 is used to perform a single 2D convolution forward pass for both encoding and decoding transformations, instead of reconstructing each crop independently in a single AEV forward step. In particular, the encoding phase is obtained exactly as explained in 2.4.1, with the only difference that the convolution is performed only on the patch area instead of the whole image. For the decoding phase, the convolution is performed starting from the volume obtained at the innermost encoding layer and decoding it back to the higher dimensional

representations of the upper layers. In the end, we will have a volume representing all the (1D) reconstructions of the crops evaluated in the patch area: we can compute each RMS error with respect to the specific crop used as input and, in the end, keep only the lowest one and compare it with the associated threshold. See Fig. 3.5 for a graphical explanation of this specific procedure.

3.5 Integration with HOG-OCR

To test the impact of this approach, we wanted to test AEVs on HOG-OCR to see if it was really possible to improve its classifying performances by further inspecting its detection.

HOG-OCR has the following stages of computations:

1. Detect individual characters throughout the image, including many FP detections;

2. Estimate the most likely locations of text lines in the image;

3. Form the most likely words, using character detections at each estimated line.

We've chosen to apply a set of AEVs right after the word creation stage, to disambiguate confusable characters. The word-search stage may produce not just a single detected string, but more than one option. In these cases, for each character or group of potential characters, a *cloud* of detections in the image space is created. Each cloud could contain one or more different characters detections, very close to each other and with a similar classifier score (which it what makes them confusable). AEVs can be applied to determine which detection is correct among the different option when the classifier score isn't accurate enough. Considering other stages in the HOG-OCR pipeline where AEVs could be applied to, the word-creation stage is the most efficient, in the sense that AEV approach only used on a few characters when the original algorithm is unable to disambiguate two or more alternatives.

Verification strategy

After the word creation stage, we dumped the detection clouds produced at line level by HOG-OCR and performed two different verification tasks:

- for each cloud with multiple candidate detections, we re-process the detection input using the corresponding AEVs and keep only the detections that gives the lowest reconstruction error;
- for each cloud with a single labeled detection, verify it using the corresponding AEV. If the reconstruction error is below the threshold, keep it as a correct detection; otherwise, try every other AEV and keep the label of one that gives the lowest reconstruction error: if it is below the the threshold, we keep it as a good detection, otherwise reject it as a FP.

3.5.1 Final results

We tried to use AEvs trained over 3 different datasets to verify the words found by HOG-OCR executed in *recognition* mode. The datasets have been completely labeled using again HOG-OCR in *validation* mode and dumping groundtruth informations for all the images in the dataset, this time after accurately selecting more characters templates in order to improve the original accuracy of the underlying classifier when needed.

***Snacks* dataset**

We first wanted to apply AEVs to HOG-OCR detections obtained for the challenging dataset we used so far throughout the whole work, here referred to as *Snacks* to distinguish it from the other datasets.

We run HOG-OCR in *recognition* mode and extracted the list of clouds generated at each image under usual condition (i.e. with the original samples used to train the classifier). Fig. 3.6 shows HOG-OCR recognition mode performance at Image level before and after applying AEV for each dataset's

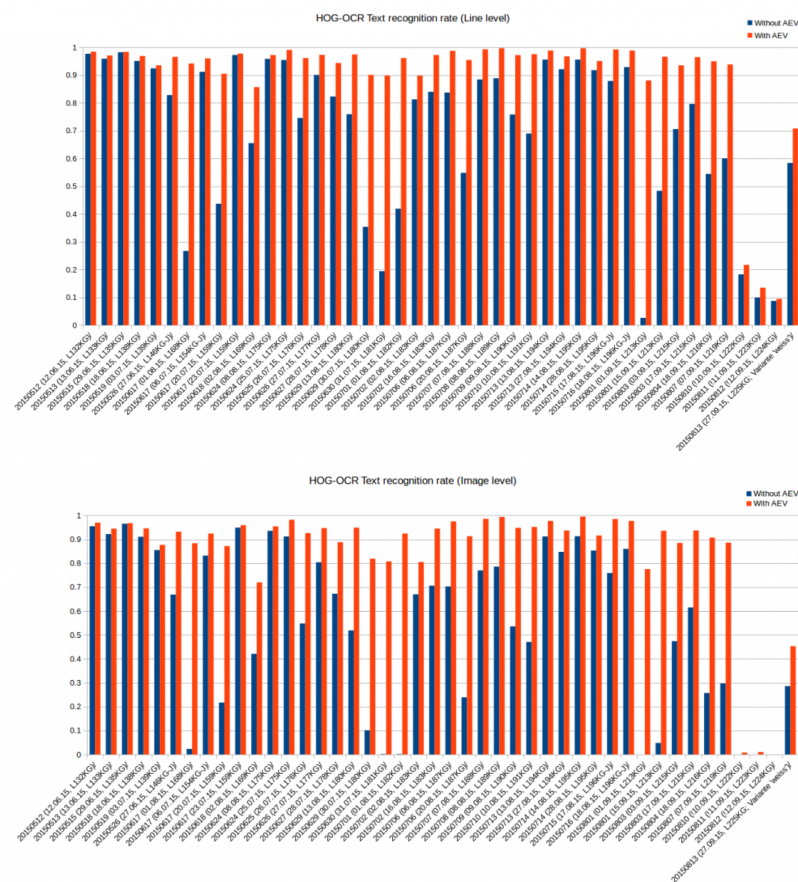

Figure 3.6: Recognition rate using HOG-OCR on *Snacks* dataset with and without AEVs application on detections. Results are referred to line level accuracy (above) and at image level (bottom).

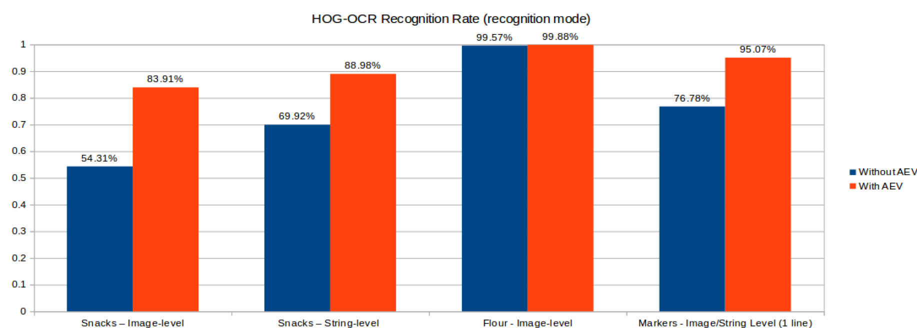

Figure 3.7: Overal recognition rate using HOG-OCR on *Snacks*, *Markers* and *Flour* complete datasets with and without AEVs application on detections. Results are referred to image level (line level is also reported for *Snacks*).

folder. It is clear that this approach increases the overall recognition rate, helping the library to better choose among confusable characters.

Other datasets

After the *Snack* experiment, we wanted to make a test also on other datasets to see if HOG-OCR could benefit of AEV strategy to improve its performances under different conditions.

We then trained more sets of AEVs to be applied to two more datasets, here referred as *Markers* and *Flour*. Due to NDA, we cannot give much informations about which company this datasets come from nor show any image, but they shows features and application requirements similar to the *Snacks* dataset. For *Markers* in particular, the recognition rate observed with HOG-OCR is above 99%, which means it presents a lower degree of variability on the real images. This case is especially interesting because we can see if AEVs could improve characters detection even when the classifier performance is already satisfying.

As Fig. 3.7 depicts, there's a clear advantage given by the application of such Verification steps. For *Snacks*, the recognition rate is increasing from 54.31% to 83.91% (+29.6%) at Image level recognition rate and from

69.92% to 88.97% (+19.05%) at Line level. For *Markers*, it is increasing from 76.78% to 95.07% accuracy (+18.29%), while for *Flour* it goes from 99.57% to 99.88% (+0.31%). The latter outcome, in particular, confirms that AEV strategy could be used for increasing classifying performances even when they are already satisfying, reaching a higher degree of accuracy by removing ambiguities when multiple characters are detected in the same position due to false activations.

Notes on timing

However, the current TensorFlow implementation of the verification step is not fast enough to compute. The timing and amount of operations for the verification step depends on these factors:

- the size of the Autoencoders (number of layers and number of neurons at each layer);
- the padding area around the original detection, used as the convolutional reconstruction input.

These parameters have to be carefully estimated with respect to the specific dataset that we want to process, in order to minimize them and keep a good verification rate. For a single reconstruction on these experiments, it requires around 0.014 s on the CPU (Intel Xeon (R) CPU E5-1620 v3 3.50 Ghz x 8) and 0.0018 s using GPU (NVIDIA GTX 1080) acceleration: considering that HOG-OCR performs a full recognition of an entire image in less than 0.01 s on most of the datasets, such AEV timings would add a consistent overhead in the computation, having to be performed several times for each multiple activation in groups of ambiguous detections. Moreover, we have to consider an additional overhead to compute the RMS error on each reconstruction resulting from the verification step: the number of RMS to compute depends on the size of the widened detection area used in the convolution step.

3.5.2 Conclusions

When launched in Recognition mode, HOG-OCR underlying classifier confuses character by assigning a higher score to inputs that matches the pattern of training templates, due to either actual similarities with non-character inputs or highly distorted/partial characters. Using AEVs as a second stage classifier, let us confuse less characters, reduce uncertainty of first stage classification output and increase the overall system accuracy.

It's easy to see the benefits of the verification step applied to an uncertain classifier for a group of activations: as long as a correct activation is present in the classifier output, AEV's verification step is likely to help select it as the correct one, if the AEVs are well trained.

Chapter 4

Future directions

4.1 Stacked Denoising Autoencoder Classifier (SDAE-C)

Showing a promising accuracy reached during training, SDAE-C architecture proved to be a robust classifier that can bring high level of generalization in classification tasks if correctly trained. Because of a non-optimal synthetic dataset and high variability of data in the scenario where it has been tested, even if it showed a good degree of digit recognition rate, the FP rate was so high that makes it unfeasible to be used in a real application. By the way, further investigation in less extreme cases should make the SDAE-C show better results and better overall accuracy.

Due of its nature, the key to obtain good performances using this classifier is to carefully build a dataset that covers most of the possible variation of samples to be recognized (which already has been proved by generating a synthetic dataset), but also some good approximation of negative/background samples order to increase the classifier accuracy and output less FPs.

4.2 Autoencoder Verifiers (AEV)

AEV approach showed some impressive results on the task of improving the best performing Datalogic OCR library on most of the challenging sets provided by their customers. This is a general approach that can be potentially applied to any classifier, providing the necessary output informations.

Since in industrial inspection there is often a stringent requirement for high accuracy, the use of an AEV can significantly improve the performance of an inspection system, as we have found with OCR, even for other kind of tasks. For example, in industrial pick-and-place applications, a robot must localize and identify one of n possible types of parts on the conveyor belt (and estimate it's pose/orientation) in order to pick it up and place it at the proper destination. As with the OCR application described above, the AEV method could be applied to avoid mis-identification of pick-and-place parts.

An ad-hoc implementation of such algorithms in pure C/C++, eventually taking advantage of careful parallelization, could improve timing performances of AEVs, and SDAE-C as well, and make it possible to include such strategies in a state-of-the-art classification pipeline.

Bibliography

[1] Ivakhnenko, A. G. (1973). Cybernetic Predicting Devices. CCM Information Corporation.

[2] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning.* MIT Press, 2016. Book in preparation for MIT Press.

[3] G. E. Hinton, R. R. Salakhutdinov. *Reducing the dimensionality of data with Neural Networks.* Science Magazine vol. 313, 2006.

[4] G. E. Hinton, S. Osindero, Y. W. Teh. *A fast learning algorithm for Deep Belief Nets*

[5] M.A. Ranzato, C. Poultney, S. Chopra, Y. LeCun. *Efficient learning of sparse representations with an energy-based model* Courant Institute of Mathematical Sciences New York University (2007)

[6] Pascal Vincent, Hugo Larochelle, Yoshua Bengio, Pierre-Antoine Manzagol. *Extracting and Composing Robust Features with Denoising Autoencoders.* Technical Report 1316, February 2008.

[7] Pascal Vincent, Hugo Larochelle, Isabelle Lajoie, Yoshua Bengio, Pierre-Antoine Manzagol. *Stacked Denoising Autoencoders: Learning Useful Representations in a Deep Network with a Local Denoising Criterion.* Journal of Machine Learning Research 11, 2016.

[8] Yoshua Bengio, Pascal Lamblin, Dan Popovici, Hugo Larochelle, et al. *Greedy layer- wise training of deep networks*. Advances in neural information processing systems, 19:153, 2007.

[9] Andrej Karpathy. Lecture notes from Stanford's course *CS231n - Convolutional Neural Networks for Visual Recognition.* http://cs231n.github.io/

[10] Sebastian Ruder. *An overview of gradient descent optimization algorithms.* http://sebastianruder.com/optimizing-gradient-descent/

[11] Michael Nielsen. *Neural Networks and Deep Learning.* (2015) http://neuralnetworksanddeeplearning.com

[12] Balázs Csanád Csáji. *Approximation with Artificial Neural Networks.* (2001) Faculty of Sciences; Eötvös Loránd University, Hungary.

[13] Bengio, Y. (2007). *Learning deep architectures for AI (Technical Report 1312).* Université de Montréal, dept. IRO.

[14] Larochelle, H., Erhan, D., Courville, A., Bergstra, J., Bengio, Y. (2007). *An empirical evaluation of deep architectures on problems with many factors of variation.* Twenty-fourth International Conference on Machine Learning (ICML'2007).

[15] Flavio Poli. *Robust string detection for industrial OCR.* Master's thesis in Software Engineering, University of Bologna (2017).

[16] Yann LeCun, Corinna Cortes, Christopher J.C. Burges. *The MNIST database of handwritten digits.* http://yann.lecun.com/exdb/mnist/

[17] Dan Cireşan, Ueli Meier, Juergen Schmidhuber. *Multi-column Deep Neural Networks for Image Classification.* CVPR 2012, p. 3642-3649

[18] X. Glorot, Y. Bengio. *Understanding the difficulty of training deep feed-forward neural networks.* DIRO, Universite de Montreal (2010) ebec, Canada

[19] Hinton, Geoffrey E.; Srivastava, Nitish; Krizhevsky, Alex; Sutskever, Ilya; Salakhutdinov, Ruslan R. *Improving neural networks by preventing co-adaptation of feature detectors* (2015)

[20] D. P. Kingma, J. Ba. *Adam: A method for stochastic optimization.* 3rd International Conference for Learning Representations, San Diego (2015)

[21] *ImageNet Large Scale Visual Recognition Challenge* website. `http://www.image-net.org/challenges/LSVRC/`

[22] Wikipedia. *Softmax function.* `https://en.wikipedia.org/wiki/Softmax_function`

[23] Wikipedia. *Rectifier (neural networks)* `https://en.wikipedia.org/wiki/Rectifier_(neural_networks)`