Masaryk University
Faculty of Informatics

Waste Sorting using Neural Networks

Bachelor's Thesis

Jozef Marko

Brno, Fall 2016

Masaryk University
Faculty of Informatics

Waste Sorting using Neural Networks

Bachelor's Thesis

Jozef Marko

Brno, Fall 2016

Replace this page with a copy of the official signed thesis assignment and the copy of the Statement of an Author.

Declaration

Hereby I declare that this paper is my original authorial work, which I have worked out on my own. All sources, references, and literature used or excerpted during elaboration of this work are properly cited and listed in complete reference to the due source.

Jozef Marko

Advisor: doc RNDr. Tomáš Brázdil, Ph.D.

Acknowledgement

I would like to thank my advisor, Tomáš Brázdil for all the help and guidance he has provided, especially for the very precise emails with feedback. I am glad to thank Procházková Alena from SAKO Brno, a.s. for organizing the excursion.

Abstract

This thesis focuses on analyzing possibilities for automatic waste sorting. For this purpose, data set of images is created in a local waste sorting company SAKO Brno, a.s.. Various Convolutional Neural Networks are used for experiments. Principal Component Analysis (PCA) applied to images and predictions made with Support Vector Machine (SVM) are considered as comparison approaches.

The thesis main goal is to automatically classify the images of waste into 7 categories: Box, Can, Blue bottle, Green bottle, White bottle, Colorful bottle and Chemical bottle.

Keywords

convolutional network, waste sorting, neural network, prediction, classification

Contents

List of Tables

List of Figures

1 Introduction

The main aim of the bachelor thesis is to design a part of the system for automatic separation of waste. For example, separating plastics from non-plastics at the recycling process. Separating different types of plastic from each other is a human labor-intensive process and so far there has been no easy solution.

The waste for separation is currently divided into two categories, dry and wet. Dry consists of plastics, glass and paper. On the other side, wet usually consists of organic material. In the system of this thesis, the focus is put on the dry waste.

There are many ways of waste sorting. Within the thesis more effective solution is trying to be found comparing to the current human-involved[1] state of waste separation. Currently human workers pick up proper raw materials (e.g. plastic bottles) situated on the belt and separate them.

The *Bachelor-thesis* system intends to optimize this process, providing additional information about the materials on the belt. The information can lead to useful statistics, in the best case used for building a mechanical system separating the materials automatically and decrease the need for human power, making the process more cost-effective.

In this thesis a neural network will be implemented which will try to classify the items from the image (image will consist of various objects for waste separation) to the proper waste separation category – e.g. plastics, cans, boxes.

The pictures are taken upright of the separation belt with the items for deep-selection, e.g. plastic bottles with some noise. Noise are the objects which are not belonging to any waste separation category, e.g. a pen on the belt with the plastic bottles.

Various neural networks algorithms for image recognition will be implemented. The algorithms will be compared against different data sets. One data set can be of various categories. The pictures might contain materials for separation as well as random objects. The goal of the thesis is to implement the system which detects the objects for separation with the satisfactory probability.

---

1. `http://www.sako.cz/stranka/cz/351/fotogalerie/`

In past years uprising of deep neural networks (DNNs) in case of classification tasks is noticed. Examples of DNNs are Convolutional Neural Netowrk or Deep Belief Network.

Convolutional neural network is a type of feed-forward artificial neural network where the individual neurons are tiled in such a way that they respond to overlapping regions in the visual field. Convolutional networks were inspired by biological processes and are variations of multilayer perceptrons designed to use minimal amounts of preprocessing, computing matrices and data structures before application [1].

Deep belief network (DBN) is a generative graphical model, or alternatively a type of deep neural network, composed of multiple layers of latent variables ("hidden units"), with connections between the layers but not between units within each layer. When trained on a set of examples in an unsupervised way, a DBN can learn to probabilistically reconstruct its inputs. The layers then act as feature detectors on inputs. After this learning step, a DBN can be further trained in a supervised way to perform classification [2].

The goal of this thesis is to compare the methods and classify waste into 7 categories: Box, Can, Blue bottle, Green bottle, White bottle, Colorful bottle and Chemical bottle. The main focus is put on prediction accuracy as well as complexity of a model. 1240 images were collected from the local waste sorting company SAKO Brno, a.s. in order to achieve this goal.

The final scenario imitating the real environment answers the question for actual application; implementation of the solution into the waste sorting companies.

2 Artificial Neural Network

2.1 Artificial Neuron

Artificial neural network consists of artificial neurons. Artificial neuron is a mathematical function modeling a biological neuron. The neuron receives one or more weighted inputs and fires an output. The dot product of input vector and vector of weights is called inner potential. On top of the potential the activation non-linear differentiable function is applied and fired as output. The inputs represent dendrites and output represents an axon within neuroscience perspective.

Weights corresponding to each input unit are denoted by vector $\vec{w} \in \mathbb{R}^n$, where w_i is the weight of the input unit x_i. The potential $p \in \mathbb{R}$ is computed as the dot product of the input vector $\vec{x} \in \mathbb{R}^n$ and the vector of weights $\vec{w}$. $|\vec{x}|$ denotes the length of the input vector.

$$p = \sum_{i=1}^{|\vec{x}|} x_i w_i \tag{2.1}$$

The activation $y \in \mathbb{R}$ is obtained after application of activation function φ on the potential p.

$$y = \varphi(p) \tag{2.2}$$

The activation functions may be of various types and shapes, among others:

- *Logistic sigmoid function* – mathematical function in S shape rising values in interval (0, 1), t being the shape parameter:

$$y(p) = \frac{1}{1 + e^{-p}} \tag{2.3}$$

- *Step function* – foundational activation function. The output is of a binary form, 1 if the input meets specific threshold θ.

$$y(p) = \begin{cases} 1 & \text{if } p \geq \theta \\ 0 & \text{if } p < \theta \end{cases} \tag{2.4}$$

- *Ramp function* – as of 2015 the most popular activation function for deep neural networks. A unit using ramp function is also called a rectified linear unit (ReLU).

$$y(p) = max(p, 0) \tag{2.5}$$

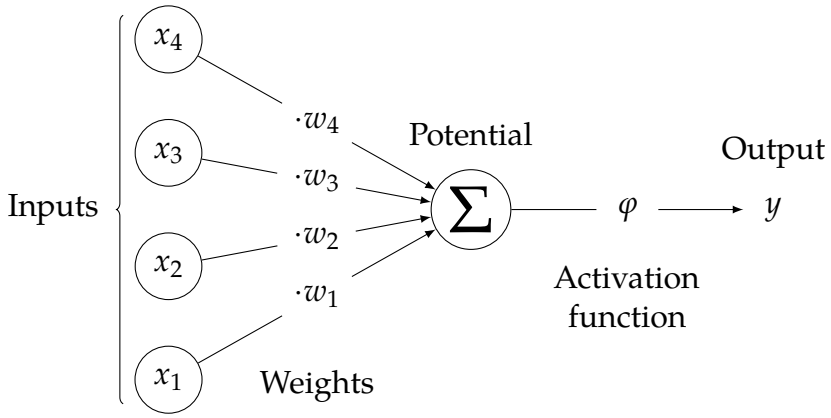

Figure 2.1: Model of an artificial neuron without bias

2.2 Feed-forward neural network

The neurons can be interconnected to form a graph. The output of a neuron is used as an input for other neurons. The acyclic such a network is called Feed-forward neural network – information flows only in one direction.

The neurons are divided into the disjoint sets, called layers $l_1, ..., l_k$. Layers $l_1, ..., l_{k-1}$ are hidden layers, l_k is an output layer. Formally layer l_0 is also considered, denoting the input data also called an input layer. The nodes in layer l_i receive as an input only the outputs of one or more connected neurons in layer l_{i-1}. Neurons in a fully connected layer have connections to all activations of neurons in the previous layer. The outputs of an input layer l_0 are the input data.

The activations of neurons in output layer l_k represent the output of the network. It may represent probabilities of belonging to classes.

The whole computation on a *feed-forward neural network* is designed with Forward propagation algorithm.

Forward propagation, the input vector is copied to the input layer. For every other layer (in topological order), firstly, the potentials are computed as multiplication of the activations of previous layer with the vector of weights of each neuron's connections. Then, the activation functions are applied to the potentials.

We denote by $y_i(\vec{x}, \vec{w})$ the value of i-th neuron in the output layer retrieved after propagation of the input $\vec{x}$ throughout the network with weights $\vec{w}$. The outputs can be represented as the class probabilities in the range $(0, 1)$ summing up to 1 using *soft-max* function. The converted probability $p_c(\vec{x}, \vec{w})$ of class c using *soft-max* function is computed as:

$$p_c(\vec{x}, \vec{w}) = \frac{e^{y_c(\vec{x},\vec{w})}}{\sum_{i=1}^{|l_k|} e^{y_i(\vec{x},\vec{w})}} \tag{2.6}$$

Shown by George Cybenko in *universal approximation theorem*, any continuous function on compact subsets of R^n can be represented by a neural network with a single hidden layer, under modest assumptions on the activation function. Training the parameters may be intricate.

The class of *feed-forward neural networks* consisting of fully connected input, hidden and output layers, usually using *sigmoid* activation function may be rephrased as *multi-layer perceptron* (MLP).

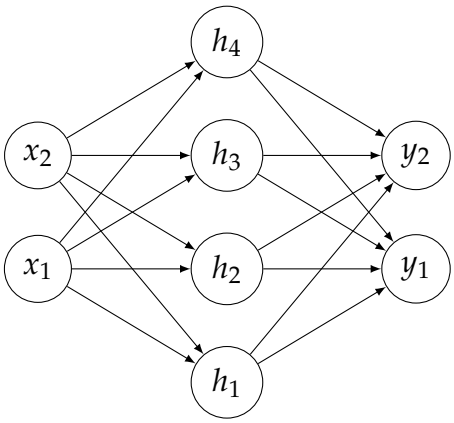

Figure 2.2: Model of a feed-forward neural network with one fully connected hidden layer and fully connected output layer

2.3 Loss function

In order to update the parameters of the network to better approximate the underlying function of the data, *loss function* is defined to compare different models.

Having the input $\vec{x}$ and output vector $\vec{y}(\vec{x}, \vec{w})$ obtained by feeding $\vec{x}$ to the input layer and propagating activations of neurons in each layer to the latter layers. Expected output vector denoted by $\vec{d_{\vec{x}}}$ where $d^c_{\vec{x}}$ denotes the binary expected value (let us assume one image can have only one class as an output) for class c holding $1 = \sum_{d^c_{\vec{x}} \in \vec{d_{\vec{x}}}} d^c_{\vec{x}}$ and $d^c_{\vec{x}} \in \{0, 1\}$. Let us denote *loss function* $e_{(\vec{x},\vec{w})}(\vec{w})$ a loss function describing an error between the computed output and expected result $\vec{d_{\vec{x}}}$. Common *loss function* of a training data point $(\vec{x}, \vec{d_{\vec{x}}})$ in classification tasks is a squared classification error:

$$e_{(\vec{x},\vec{d_{\vec{x}}})}(\vec{w}) = \frac{1}{2} \sum_{c=1}^{|l_k|} (y_c(\vec{x}, \vec{w}) - d^c_{\vec{x}})^2 \tag{2.7}$$

Loss function $e_{X_i}(\vec{w})$ over subset X_i of input data set X is defined as the mean error over all data points in X_i.

$$e_{X_i}(\vec{w}) = \frac{1}{|X_i|} \sum_{(\vec{x},\vec{d_{\vec{x}}}) \in X_i} e_{(\vec{x},\vec{d_{\vec{x}}})}(\vec{w}) \tag{2.8}$$

Cross entropy

The classification error nor squared classification error have been proven to show the best results in classification tasks. The cross entropy method puts no importance on activations of output neurons of wrong labels and achieves better results in over-all model robustness and precision.

In cross entropy method, firstly, the activations $\vec{y}(\vec{x}, \vec{w})$ in output layer are transformed into probablities as described in section 2.2. The cross entropy $ce_{(\vec{x},\vec{d_{\vec{x}}})}(\vec{w})$ of one data example $(\vec{x}, \vec{d_{\vec{x}}})$ is then computed as a natural logarithm of a probability of a correct class:

$$ce_{(\vec{x},\vec{d_{\vec{x}}})}(\vec{w}) = \sum_{c=1}^{|l_k|} ln(p_c(\vec{x}, \vec{w})) \cdot d^c_{\vec{x}} \tag{2.9}$$

The cross entropy of a set of data points is calculated as described in figure 2.8.

2.4 Gradient descent

The neural network is differentiable and on the the outputs are applied differentiable functions, either cross entropy or mean squared error. Thus the loss functions defined in section 2.3 are differentiable.

In order to minimize the loss function, the learning algorithm called Gradient descent with learning rate η in (0,1) was introduced.

1. The input layer of the neural network is fed with input data, then forward propagated to the output layer. The gradient of chosen loss function $e_{(\vec{x},\vec{d_{\vec{x}}})}(\vec{w})$ is calculated afterwards.

2. **Weight update**
$$\vec{w'} = \vec{w} - \eta \nabla e_{(\vec{x},\vec{d_{\vec{x}}})}(\vec{w}) \tag{2.10}$$

Gradient $\nabla e_{(\vec{x},\vec{d_{\vec{x}}})}(\vec{w})$ points to a direction from $\vec{w}$ in space W with the steepest increase on value of $e_{(\vec{x},\vec{d_{\vec{x}}})}(\vec{w})$. Thus intuitively updating the weight vector $\vec{w}$ within the gradient descent algorithm in opposite direction of the $\nabla e_{(\vec{x},\vec{d_{\vec{x}}})}(\vec{w})$ to $\vec{w'}$ should be a good heuristic for finding a local minimum of $e_{(\vec{x},\vec{d_{\vec{x}}})}(\vec{w})$. However, nothing is guaranteed.

Stochastic gradient descent (SGD) is known as incremental gradient descent, weights updating is done in iterations, after each input example. Many weight updates of single data examples proved to be solid solution in finding the minima of $e_X(\vec{w})$.

In reflection with *Batch gradient descent* (BGD) where the losses are averaged over a batch of training data points to $e_{X_i}(\vec{w})$, following with back-propagation of $e_{X_i}(\vec{w})$ to calculate the gradient $\nabla e_{X_i}(\vec{w})$ reflecting the whole batch. Weight updating as described in point 2. is done once after each batch.

Data splitting

Usually the data are split into training, validation and test data sets. The training data set is used for tuning the network's configuration.

The validation data set is used to avoid over-fitting. When the error on validation data set, called validation loss, stops to decrease while training loss still decreases, the network is starting to recognize specific patterns occurring only in the training data set. Often that is the moment when the training should finish.

The unseen test data set is used after final model was defined to check how well the network is performing in real environment.

2.5 Back-propagation

The only missing part is to describe a way how to calculate a gradient of a loss function on a feed-forward neural network. Smart algorithm computing the gradient of a complicated loss function depending on many variables is called Backward propagation of errors or Back-propagation.

Let there be a *feed-forward* neural network with L layers stacked in topological order with activation function φ and loss function $e_{(\vec{x},\vec{d}_{\vec{x}})}(\vec{w})$. l_k denotes the output layer and l_0 the input layer. Matrix of neurons activations, y_i^m represents the activation of i-th neuron in layer l_m. Matrix of neurons potentials, p_i^m represents the potential of i-th neuron in layer l_m. The connections matrix W, where w_{ij}^m is the weight on the connection between i-th neuron in the layer l_{m-1} and j-th neuron in the layer l_m. Back-propagation algorithm is defined in following steps:

1. Feed-forward propagation as described in section 2.2.

2. **Output delta vector**, calculation of the difference between expected $\vec{d}_{\vec{x}}$ and actual output vector $\vec{y}^k$. Output delta vector of squared classification error loss function is computed as shown on the figure below, where y_c^k denotes the activation of c-th neuron in the output layer l_k, the probability of the data point to belong to the class c respectively.

$$\frac{\partial e}{\partial y_c^k} = y_c^k - d_{\vec{x}}^c \tag{2.11}$$

   Being the partial derivative of the loss function.

3. **Backward propagation**, backwards calculation of the derivatives of activations of neurons in hidden layers, starting from the last hidden layer ($m = k - 1$) to the first one ($m = 1$).

$$\frac{\partial e}{\partial y_i^m} = \sum_{j=1}^{|l_m|} \frac{\partial e}{\partial y_j^{m+1}} \cdot \varphi'(p_j^{m+1}) \cdot w_{ij}^{m+1} \tag{2.12}$$

$\frac{\partial e}{\partial y_j^{m+1}}$ is already known at the time of calculation.

4. $\nabla e_{(\vec{x},\vec{d}_{\vec{x}})}(\vec{w})$ **calculation**, for each m; 1 <= m <= L:

$$\frac{\partial e_{(\vec{x},\vec{d}_{\vec{x}})}(\vec{w})}{\partial w_{ij}^m} = \frac{\partial e_{(\vec{x},\vec{d}_{\vec{x}})}(\vec{w})}{\partial y_j^m} \cdot \varphi'(p_j^m) \cdot y_i^{m-1} \tag{2.13}$$

2.6 Momentum, learning rate decay, dropout and L2 regularization

Since the *loss function* depends on many variables, the landscape of *loss function* has many critical points (maxima, minima, and saddle points), thus SGD with specific parameters (learning rate, momentum) can stuck within one local minimum (e.g. learning rate η^k is too small to leave the local minimum). The local minimum can be arbitrarily far away from the global minimum. Many techniques have been introduced to deal with the issue. The ones used within the thesis:

- **Momentum** – gradient descent with momentum remembers previous weight update $\Delta\vec{w}$. Adds the current weight update with α of the previous weight update; α is the momentum parameter. Using momentum can speed up the learning process and reach better accuracy.

$$\Delta\vec{w} = \alpha\Delta\vec{w} - \eta\nabla e(\vec{w}) \tag{2.14}$$

$$\vec{w} = \vec{w} + \Delta\vec{w} \tag{2.15}$$

- **Learning rate decay** – the latter part of the training process is rather sensitive on high learning rate. Learning rate decay

constant β is set; $0 < \beta < 1$. In order to tune up the network, after each epoch (all training data examples were fed to the network) the new learning rate η^{k+1} is obtained by β-decaying the learning rate η^k used within epoch number k. Where η^1 is equal to a starting learning rate value η used in first epoch.

$$\eta^{k+1} = \beta \cdot \eta^k \tag{2.16}$$

Alternatively:

$$\eta^{k+1} = \beta^k \cdot \eta \tag{2.17}$$

- **Dropout** – is a regularization technique preventing over-fitting (good performance on training data, bad on validation data). With a probability of θ – the neuron (either visible or hidden) is dropped out of the training batch processing. If the neuron is dropped out, its activation is always 0, also the weights on the neuron's connections are not updated. Retained neuron's activation is scaled up by $\frac{1}{\theta}$ so the expected sum holds. The network's evaluation is not dependent on specific neurons, making the model more complex, discovering robust features which better generalize to new data.

- **L2 regularization** – compute squared magnitude of all parameters (weights) and add it as a penalization to the loss function. L2 regularization encourages the neurons to use all of their inputs rather than just a few a lot. It penalizes peaky weight vectors, prefers diffuse ones [3].

$$e = e + \phi \cdot \sum_{w_{ij}^k \in W} (w_{ij}^k)^2 \tag{2.18}$$

3 Deep Neural Networks

3.1 Convolutional Neural Network

Convolutional neural network (CNN) is a feed-forward artificial neural network in which the organization of neurons is similar to the animal visual cortex. In order to recognize the shape of an object, the local arrangement of pixels is important. CNN starts with recognition of smaller local patterns on the image and concatenate them into more complex shapes. CNN was proved to be efficient especially in object recognition on an image. CNNs might be an effective solution to the waste sorting problem.

CNN explicitly assumes the input is an image and reflects it onto its architecture. CNN usually contains Convolutional layer, Pooling layer and Fully-connected layer. Convolutional layers and Pooling layers are stacked on each other, fully-connected layers at the top of the network outputs the class probabilities.

3.1.1 Convolutional layer

Convolutional layer consists of neurons connected to a small region of pixels (also called the receptive field) of previous layer. The neurons in same feature map share the same weights [4].

Filter

Convolutional layer (CL) contains a set of learnable filters. One filter activates when a specific shape or blob of color occurs within a local area [5]. Each CL has multiple filters F. *Filter* $f_i \in F$ is a set of learnable weights corresponding to the neurons in previous layer. Filter is small spatially (along width and height) and extends along the full depth c of the previous layer.

Filter's size is denoted by (f_w, f_h)[1]. Therefore one filter adds r_w x r_h x c parameters, other weights are not considered.

CL has a depth D if it recognizes D various features in the image using D various learnable filters f_1, ..., f_d. Thus, CL contains r_w x r_h x

1. Other indices are used to denote specific filter.

c x d learnable weights. Every neuron in CL uses weights of exactly one CL's filter, many neurons use the same weights.

The neurons in CL are segmented into feature maps by the filter they are using. Neurons belonging to feature map fm_i share the same weight matrix of filter f_i. CL forms D feature maps.

Usual filter sizes are 3x3, 5x5 or less frequently 7x7.

Receptive field

Let us define a position (x, y, c) of a neuron in input layer (image); x and y describe the location of the pixel in the image. The c component denotes specific channel of the pixel. In general the length of the third dimension is rephrased as depth.

Receptive field r_n of a neuron n in CL specifies the local area of neuron's connectivity onto the previous layer. All neurons within the same CL have the same size of their *receptive fields*. A neuron in convolutional layer with *filter size* (f_w, f_h) will have f_w x f_h x c connections, where c stands for the depth of the previous layer.

Without loss of generality, let us assume each neuron in previous layer has assigned a position as a 3-dimensional tuple. Let there be any two arbitrary neurons in the previous layer with positions (x_1, y_1, z_1) and (x_2, y_2, z_2) connected to the same neuron n in the convolutional layer. Then following holds:

$$abs(x_1 - x_2) < r_w \tag{3.1}$$

$$abs(y_1 - y_2) < r_h \tag{3.2}$$

Where $abs(x)$ stands for the absolute value of x.

Furthermore, if neuron with position (x, y, z) in previous layer is connected to the neuron n in CL then all neurons in previous layer with first two co-ordinates equal to (x, y) are connected to n.

All connections of neuron n form the receptive field r_n of a neuron n. The volume of the receptive field is always equal to the filter size of a particular CL multiplied by the depth of previous layer.

Let there be a set of positions X_n of neurons belonging to the receptive field r_n of neuron n in CL. The position of a neuron n using weights $\vec{w}_n$ is defined as a tuple (x, y, c) such as:

$$\forall (x_i, y_i, z_i) \in X_n : 0 < x_i - x < f_w; 0 < y_i - y < f_h; \tag{3.3}$$

$$\vec{w}_n = \vec{w}_{f_c} \tag{3.4}$$

The weights used for convolution belong to the filter f_c, thus the depth of CL is equal to D.

Activation of neuron

Let there be a neuron n in CL at position (x_i, y_j, k) recognizing filter $f_k \in F$. Let us denote the weight of the bottom-left input of filter f_k with various depth co-ordinate c as $w^k_{(0,0,c)}$ and $w^k_{(r_w-1,r_h-1,c)}$ as the top-rights. Let there be b_k the bias of filter f_k. The depth of the previous layer being C, the potential $p_{(x_i,y_j,f_k)}$ of the n is computed as a dot product between the filter's weights $\vec{w}^k$ and the receptive field of the neuron r_n [1]:

$$p_{(x_i,y_j,f_k)} = b_k + \sum_{l=0}^{f_w-1} \sum_{m=0}^{f_h-1} \sum_{c=1}^{C} w^k_{(l,m,o)} \cdot v(x_i + l, y_j + m, o) \tag{3.5}$$

Where the $v(x, y, c)$ represents the activation of neuron in previous layer at position (x, y, c). The process for one feature map can be seen in Figure 3.1

The activation is computed by application of the activation function over the potential. As an activation function, most of the time, *ramp function* is used, also referred as the ReLU unit.

If the specific feature matters in one part of the image, usually the feature is important also in the rest of the image. That is the intuition why weights of one filter applies over the whole feature map.

Stride

One of the most important hyper-parameters next to the number of filters is *stride*. If stride (s_w, s_h) is applied on the convolutional layer then for any two neurons with positions (x_i, y_i, c), (x_j, y_j, c) belonging to the same filter f_c applies:

$$abs(x_i - x_j) \geq s_w \tag{3.6}$$

$$abs(y_i - y_j) \geq s_h \tag{3.7}$$

If stride is set to (s_w, s_h), the CL positioned after layer with shape (l_w, l_h, l_D) will have t x g x D neurons.

$$t = \frac{(l_w - r_w + mod(r_w, 2))}{s_w} \tag{3.8}$$

$$g = \frac{l_h - r_h + mod(r_h, 2)}{s_h} \tag{3.9}$$

Where mod(x,y) computes the remainder of x divided by y. If numerator is not divisible by denominator the *stride* must be changed or *zero-padding* (see below) should be used.

Zero-padding

Zero-padding is one of the last hyper-parameters. Sometimes, it might be convenient to pad the input around the borders with zero-pixels, pixels having 0 in all channels, in order to keep a specific input size, or to normalize images with various shapes. Usually, if zero-padding is set to Z, Z zero-columns and zero-rows are added around the input volume.

3.1.2 Pooling layer

Pooling layer (PL) is an effective way of non-linear down-sampling. It has as the convolutional layer receptive field and *stride*, however is not adding any learnable paramaters. PL layer is usually put after the CL.

The receptive field r_n of neuron n in PL is 2 dimensional (Figure 3.2). It extends over a square of neurons in one feature map (or the neurons sharing the same third coordinate in case of not CL layer). Let us denote $width^{l-1}$ and h^{l-1} the spatial width and height of the previous layer. For each feature map there is P neurons in PL.

$$P = \frac{width^{l-1} \cdot h^{l-1}}{s_w \cdot s_h} \tag{3.10}$$

Max-pooling layer is the most frequently used pooling layer. Each neuron outputs the maximum of its receptive field. Usually the *stride* is the same as the size of receptive field. The receptive fields do not

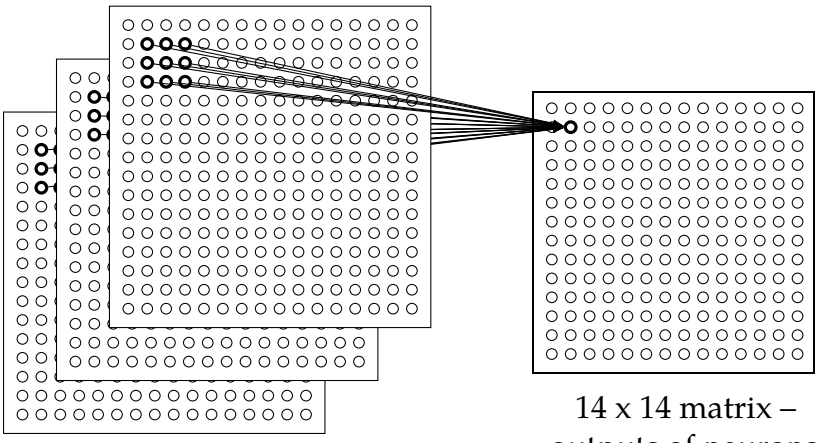

Figure 3.1: Model of convolution. Neurons on the right represent one feature map of convolutional layer. Activations represent convolutions of receptive field 3 x 3 (x 3 – along all depth of the input).

overlap, but touch. In most cases *stride* and size of *receptive field* are 2 x 2 (Figure 3.2).

The output $y(x_i, y_j, f_k)$ of the max-pooling neuron n with position (x_n, y_n, z_n) and receptive field r_n of size f_w x f_h in PL with *stride* (s_w, s_h):

$$y(x_n, y_n, z_n) = \max_{0 \leq l < f_w} \max_{0 \leq m < f_h} v(x_n + l, y_j + m, f_k) \tag{3.11}$$

Inspired by a processes in the visual cortex of animals MAX-pooling layer amplifies the most present feature (pattern) of its receptive field and throws away rest. The intuition is that once a feature has been found, its rough location relative to other features is more important than its exact location. The pooling layer is effectively reducing the spatial size of the representation, does not add any new parameters – reducing them for latter layers, making the computation more feasible.

The idea of pooling layer was created back in the time with lack of computational power. Due to its destructiveness – throwing away 75% of input information in case of small 2 x 2 receptive field, the current trend prefers stacked convolutional layers eventually with *stride* and uses pooling layers very occasionally or discards them altogether [6].

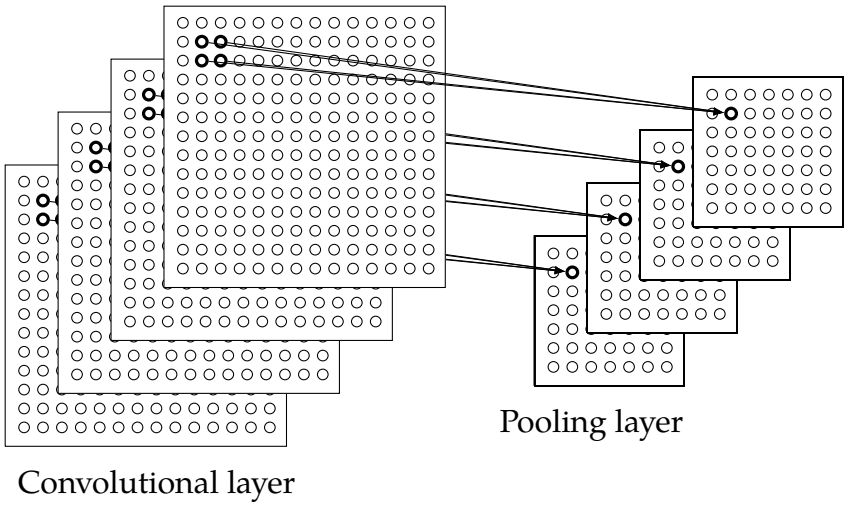

Figure 3.2: Model of pooling. The convolutional layer on the left has feature maps of four filters. Pooling layer down-samples the feature maps over x and y axis. However, it keeps the depth.

3.1.3 Structure

As a feed-forward artificial neural network, the CNN consists of neurons with learnable weights and biases. CNN's neurons still contains activation function and the whole network expresses single differentiable score function. The position of the pixel matters in comparison with MLP. It receives 3 dimensional space input (x, y, z) – the value of z-th channel of the pixel or occurrence of z-th feature of CL at position (x, y). One pixel is usually made of three channels – red, green and blue.

The convolutional layer and pooling layer are locally connected to the outputs of the previous layer, recognizing or magnifying local patterns in the image. Pooling layer is usually put after the convolutional layer. This pair of layers is repeatedly stacked upon each other following with the fully connected layers at the top.

The positions of neurons in previous layer are always considered without *stride* as inputs for latter layers. If in layer l_{m-1} with stride

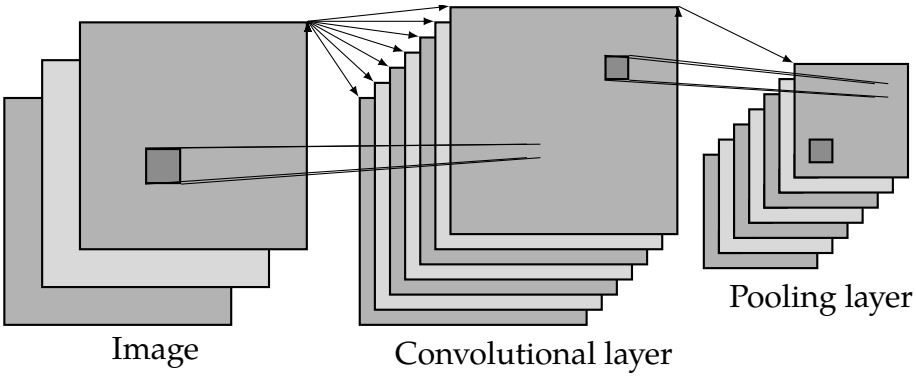

Figure 3.3: Image (16x16) as IL is locally analyzed by 7 filters (3x3) in CL forming output of shape 7x14x14. PL takes local maximum and outputs matrix of shape 7x7x7. Further 7x2x2 can be used as a receptive field of latter CL.

(s_w^{m-1}, s_h^{m-1}) there are neurons at positions:

$$\{(0,0,0),(0,0,1),(s_w^{m-1},0,0),(s_w^{m-1},0,1),(0,s_h^{m-1},0),(0,s_h^{m-1},1),...\} \tag{3.12}$$

then we can consider them to be at positions:

$$\{(0,0,0),(0,0,1),(1,0,0),(1,0,1),(0,1,0),(0,1,1),...\} \tag{3.13}$$

Usual architecture can be: input layer(IL), CL, PL, CL, PL, full-connected layer (FC), FC (Figure 3.3 shows IL, CL and PL stacked together). Recent studies suggest stacking many CLs together with fewer PLs.

The fully connected layer is connected to all outputs of last pooling layer. The outputs of last pooling layer should already represent complex structures and shapes. The fully connected layer follows usually with another one or two layers finally outputting the class scores.

3.1.4 Back-propagation

The single evaluation is completely consistent with the feed-forward neural network. The input data or activations are passed to next layers, dot product is computed over which activation function is applied. Down-sampling the network using pooling layer might be present. At

the end two or three fully connected layers are stacked. Cross-entropy or mean squared error as loss function are usually used as defined in section 2.4. In order to use gradient descent learning algorithm, the gradient must be computed.

The usual back-propagation algorithm as defined in 2.5 is applied with two technical updates. Classical back-propagation algorithm would calculate different partial derivatives of weights belonging to the neurons in same filter, however these must stay the same. Therefore, derivatives of loss function with respect to weights of neurons belonging to the same feature map are added up together.

Let there be w^k_{ijc} the weight of the c-th channel of the input at relative position (i, j) of filter f_k in CL l_m of shape t x g x D. Let denote by $y^m_{x,y,z}$ the activation of neuron in layer l_m at position (x, y, z) and by $p^m_{x,y,z}$ the potential of neuron in layer l_m at position (x, y, z). The gradient of a loss function after feed-forwarding data point example $(\vec{x}, \vec{d})$ throughout the network is calculated as follows:

$$\frac{\partial e_{(\vec{x},\vec{d})}(\vec{w})}{\partial w^k_{ijc}} = \sum_{x=1}^{t} \sum_{y=1}^{g} \frac{\partial e_{(\vec{x},\vec{d})}(\vec{w})}{\partial y^m_{x \cdot s_w, y \cdot s_h, k}} \cdot \varphi'(p^m_{x \cdot s_w, y \cdot s_h, k}) \cdot y^{m-1}_{x \cdot s_w + i, y \cdot s_h + j, c} \quad (3.14)$$

All the derivatives on the right side of the equation are known from the back-propagation described in section 2.5.

The update of back-propagation itself is when dealing with max-pooling layers. The back-propagating error is routed only to those neurons which have not been filtered with max-pooling. It is usual to track indicies of kept neurons during forward propagation to speed up the back-propagation.

3.2 Deep Autoencoder

3.2.1 Autoencoder

Autoencoder is a feed-forward neural network where expected output is equal to the input of the network – its goal is to reconstruct its own inputs. Therefore, autoencoders are belonging to the group of unsupervised learning models [4].

Usually autoencoder consists of an input layer l_0, one or many hidden layers l_1,...,l_{k-1} and output layer l_k, such that following holds:

$$|l_0| > |l_1| > ... > |l_c| < ... < |l_{k-1}| < |l_k| \tag{3.15}$$

$$|l_0| = |l_k| \tag{3.16}$$

Since the idea of autoencoders is very similar to Restricted-Boltzman Machine, it is common for the structure of autoencoders to follow the rule:

$$|l_i| = |l_{k-i}| \tag{3.17}$$

Let us denote the layer l_c, such that the number of neurons in l_c is lower than in any other layer. The l_c is an encoding layer. The feed-forward neural network consisting of layers $l_i; i <= c$, is called *encoder*. Expectedly, stacked layers l_j, such as $j >= c$ is called *decoder*. Eeach autoencoder consists of *encoder* and *decoder*.

The mean squared error loss function on data example set X:

$$e_X(\vec{w}) = \frac{1}{|X|} \sum_{\vec{x} \in X} \sum_{x_i \in \vec{x}} (x_i - (decode \circ encode)(x_i))^2 \tag{3.18}$$

The *encoder* can be used for compression. Unlike Principal Component Analysis analysis restricted to linear mapping, the *encoder* represents non-linear richer underlying structures of the data [7]. The activations of the l_c layer can be further used for classification. Fully-connected layers are appended with the size of the last corresponding to the number of labels. Usual learning algorithms are used.

3.2.2 Deep Autoencoder

Deep Autoencoder consists of many layers stacked on each other allowing to discover more complicated and non-linear structures of the data. Since it may be complicated to tune deep autoencoder network, commonly the training procedure is made of two steps:

1. **Pre-training**, each layer 1_1,...,l_c is pre-trained. Firstly the pair l_0 as an example and l_1 as encoder is used. The goal is to find representation of l_0 in l_1 using the right optimizer. The weights l_0 to l_1 and l_1 to l_0 may be tied up representing Restricted Boltzman-Machine. When good representation of l_0 inputs is encoded in l_1 the pair l_1, l_2 is pre-trained further till pair l_{c-1}, l_c is reached.

2. **Fine-tuning**, the full network is connected and fine-tuned. In case of classification, the encodings of input data points can be used for classification training or the whole network l_0 -> ... -> l_c -> fc_1 -> fc_2 is part of the supervised learning.

4 Implementation

4.1 Data Set

In order to satisfy the main aim of the thesis, the data set of waste materials had to be created. More than 1,500 pictures have been taken in local waste sorting company SAKO Brno, a.s. The images were further categorized and filtered to create a data set of 1,240 data points. Each picture consists of only one easily identified main object. The object belongs to one of following class:

- **Blue** – ordinary blue plastic PET bottle
- **Box** – box of milk or juice
- **Can** – can of beer or energy drink, usually a smaller object comparing to other categories
- **Chemical** - chemical bottle of various cleaning liquid chemicals, coming from more resilient plastics. Chemical bottle can be made of all possible colors, making it one of the toughest categories to classify.
- **Colorful** - ordinary colorful (not blue, green or white) plastic PET bottle
- **Green** - ordinary green plastic PET bottle
- **White** - ordinary white plastic PET bottle

Since the frequency of various classes naturally differs, the sizes of classes in data set are not the same. The exact magnitudes of specific categories can be seen in Table 4.1.

In need for better understanding of the results, the data were split into various data sets. Each data set can be characterized by following properties:

- **Type** – examples can be seen in Figure 4.1
  - **Original**: the original scaled images taken in the waste sorting company.

Figure 4.1: The 48x36 original images, 32x32 cropped & bordered and 32x32 cropped & extended versions

  - **Cropped & Bordered**: the main object for categorization is cropped out of the image. Since the cuts are of different sizes – the images are normalized to form the same shape. Let there be a cropped image of shape (x, y) and chosen final square resolution (s, s). Firstly, the ratio r is counted:

    $$r = s/max(x, y) \tag{4.1}$$

    The image is scaled to shape $(x * r, y * r)$. Without loss of generality, let us assume $x * r = s$, therefore $y * r \leq s$. Black stripes of width w; $w = \frac{s-y*r}{2}$, are appended along y-axis of the cropped images to form a square.
  - **Cropped & Extended**: The cropping is done in the same way as in Cropped & Border type. The image itself is used to tile the space around the image to form the graph instead of black stripes.

- **Resolution** – images are scaled into various shapes to fit the predictor. Usually in the thesis resolution 32x32 is preferred, but also other resolutions such as 16x16, 48x48 or higher were

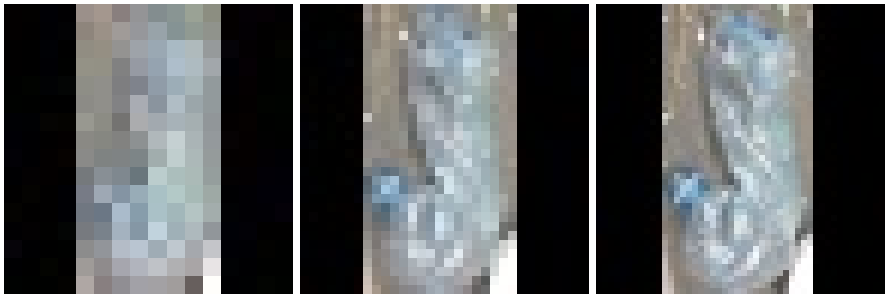

Figure 4.2: Blue bottle of resolutions: 16x16, 32x32 and 48x48

included in the tests. Various resolutions are shown in Figure 4.2.

- **Difficulty** – manually labeled as part of the thesis
  - **Hard**: images labeled as hard are also used in the data set. Hard images usually consists of blurred images, images with destroyed main object or images with more objects on the plane. Examples of images labeled as hard are introduced in Figure 4.3 and numbers of hard images are described in Figure 4.1.
  - **Normal**: data set consists only of images not labeled as hard.

- **Extended data set** $\in$ {True, False} – extended data set is also containing the pictures which where created by random transformations of original images (such as changes of rotation, contrast, brightness). Extended data set is 8 times larger (9,920) and effectively fights over-fitting.

- **Simplified categories** $\in$ {True, False} – if simplified categorization is set on, the mapping of categories described in Table 4.2 applies.

Figure 4.3: Blue bottle (64x48) at top-left, white bottle at bottom-left, green bottle at top-right and chemical bottle at bottom-right. All of the examples have been labeled as hard – either the background is messy or contains objects belonging to different categories or the bottle is distorted. Images shown in Figure 4.1 and Figure 4.2 have not been categorized as hard.

Category	Blue	Box	Can	Chem.	Color.	Green	White	Sum
Hard	57	38	35	26	32	106	97	391
Normal	149	139	76	93	57	172	163	849
Sum	206	177	111	119	89	278	260	1240

Table 4.1: Sizes of various classes

Orig.	Blue	Box	Can	Chemical	Colorful	Green	White
Simp.	Plastics	Box	Can	Plastics	Plastics	Plastics	Plastics

Table 4.2: Original to simplified mapping of categories

4.2 Project

As part of the thesis the project *Bachelor-thesis*[1] was developed in *Python*[2].

4.2.1 Libraries

Bachelor-thesis uses various libraries, among others the most important are:

1. **NumPy**[3] – library making Python effective enough for scientific projects. It effectively stores and transforms N-dimensional arrays.

2. **TensorFlow**[4] – open source library for Machine Learning. It allows to deploy the computations to multiple CPUs.

3. **Keras**[5] – is a high-level neural network library build on top of TensorFlow.

4. **scikit-learn**[6] – simple tool for data mining and data analysis in Python. However, deploying neural networks on scikit-learn was much slower than on TensorFlow.

4.2.2 Structure

Bachelor-thesis is split into following independent parts, folders respectively[7]:

- Images – contains scaled raw images of all types: Cropped & Bordered, Cropped & Extended and Original.

---

1. The project is attached to the thesis or can be cloned from: `https://github.com/jodik/Bachelor-thesis`
2. Programming language see: `https://www.python.org/`
3. `http://www.numpy.org/`
4. `https://www.tensorflow.org/`
5. `https://keras.io/`
6. `http://scikit-learn.org/stable/`
7. The root folder of the project is *Bachelor-thesis/*

- Datasets – contains raw byte data of data sets consisting of data.byte (scaled images and images created by transformation), ishard.byte (information whether particular image is classified as hard), labels.byte, names.byte (containing original names of the image files)

- Programming

  - Analysis – scripts handling the analysis and export of the results.
  - DataScripts – data scripts containing classes *DataSet* and *DataSets*. Files handling the loading and processing of the data situated in Datasets/ folder are also situated in the folder.
  - HelperScripts – global functions used over the whole project.
  - Learning – the main folder containing all scripts responsible for machine learning part: CNN's, PCA & SVM's as well as Deep Autoencoder's scripts.
  - Preprocessing – contains programs responding for scaling the images, enlarging the data set using transformations and creating byte data files.
  - configuration.py – global configuration of the project such as the definition of data set.
  - main.py – starting point of the project.

4.3 Preprocessing

The program *scaling_script.py* is capable of scaling all images into particular resolution and store them correctly to *Images/*.

The code in file *normalizingImages.py* normalizes cropped images into the squares. Either by copying their content to form a square and creating Cropped & Extended data set or by appending black stripes around the image and creating Cropped & Bordered data set.

creatingByteDataset.py is the most important script of preprocessing. It generates new images by applying transformations on them

(change of brightness, saturation, contrast and flip of the image vertically and/or horizontally) and forms Extended data set (Section 4.1). The result is saved into byte files.

For transformation functions from TensorFlow library are used. The whole procedure of generation is very time consuming.

4.4 DataScripts

Very important part of the project responsible for reading, processing and normalizing the data.

Class *DataSet* is designed to store all information about the data set: images, labels, names, "ishard" (which images were labeled as hard), edge descriptors and original images (not normalized). It can also apply transformation over the data, such as permute them. Class *DataSets* stores training, validation and test *DataSet*.

File *data_reader.py* extracts the data from byte files and filters unselected categories in configuration.py file.

File *data_process.py* splits the data into subsets according to the configuration.py and filters out the images in validation or test set which were created by transformation of the image occurring in training data set.

In file *data_normalization.py* data are normalized to mean 0 and standard deviation 1. Also, subsets are normalized over categories count (random images are copied), thus volume of each class within the *DataSet* is the same.

4.5 Learning

Learning is the core of *Bachelor-thesis* containing all learning scripts. The most crucial sub-parts are:

- *Autoencoder/*
- *CNN/* – all programs around CNNs and their results
  - *model.py* – puts together a TensorFlow model corresponding to *CNN1* (Section 5.2.1)

  - *cnn_default.py* – creates a TensorFlow model defined in model.py, loads its data and performs learning.
  - *cnn_default_deep_3.py* – extends model defined in model.py to reflect the architecture of *CNN2* (Section 5.2.2). Extends class CNNDefault defined in cnn_default.py, loads the data and performs learning with proper configuration.
  - configuration_*.py – defines all hyper-parameters of particular CNN's architecture. This file is logged together with running history and results if variable *WRITE_TO_FILE* in *Bachelor-thesis/Programming/configuration.py* is set to True.

- *PCA_SVM/*
  - *pca_svm.py* – contains necessary code using *scikit-learn* library to preform PCA & SVM class predictions. It implements the same interface as scripts in *CNN/* and *Autoencoder/*: the constructor takes data sets as parameter and the class implements method *run* which returns validation error and validation confusion matrix.
  - *configuration_default.py* – definition of gamma, c, number of components and kernel function hyper-parameters.

5 Results

Various models have been evaluated on different Cropped data sets to find the best approach suiting the task. Then the best model on Original data was evaluated as part of the real environment simulation. All experiments use normalized data (equal volumes of images in classes, the mean equals to 0 and standard deviation equals to 1).

The data were split into training, validation and test data sets. Training data set was used for training the model, validation data set provided a hint when the learning algorithm stops to generalize and the training should be finished.

In most cases, test data set had never been used until the finalization of the thesis. The best performing model was evaluated on the test data and proclaimed as the result.

K-Fold Cross Validation

K-fold cross validation is a method estimating the performance of a model. The data are split into k folds F_1 to F_k. There is k testing iterations. In each iteration different fold F_i is considered as validation data and the rest is used as training data. The average validation result over all iterations is the model performance.

This approach is more complicated when conducting model selection as well. Ideally, the data would be split into two halves the test and training data. Within the training data the k-cross validation would be conducted to choose the best model. The winning model's performance would be estimated using k-cross validation method on the test data.

1240 original data points were not enough for such an approach. Therefore, within the thesis' experiments the data points were permuted randomly. 20% of the data were selected as the test data. The rest 80% of the data were available for model selection using k-cross validation. Finally, to test the best model on the test data following approach was chosen: k-cross validation was executed on the training data, at the end of each iteration the model was tested on the test data set and the results were averaged.

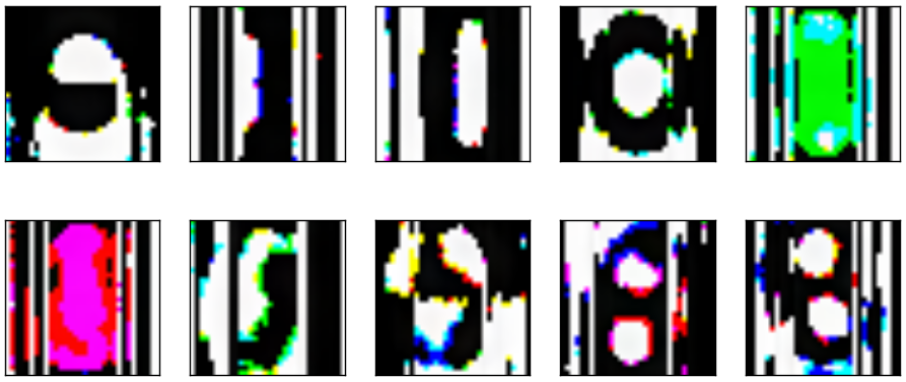

Figure 5.1: 10 out of 50 components discovered by PCA.

5.1 PCA & SVM

PCA [8] & SVM [9] is very simple model where PCA algorithm is applied on the images (Figure 5.1). From each image the vector of its components found by PCA is put as an input to SVM.

This model is beneficial when comparing simple linear or almost linear approach to more complex non-linear neural networks. SVM should not outperform neural networks.

As described in k-fold crossvalidation, training data set was split into validation subset and training subset. The training subset was used for PCA analysis. Then, data set was transformed into vector of particular occurrences of components found by PCA. SVM model was trained against the transformed training subset.

In training, the parameter grid search algorithm was used to gradually find the best hyper-parameters of SVM on the cross-validation's training data set. Best performing SVM predictor was evaluated on transformed validation data fold. The parameter grid was adjusted accordingly to the new knowledge and the whole process repeated until satisfactory results have been found.

Overall PCA & SVM provided relatively easy to train solution with fast prediction times (running only on one thread) with decent results as shown in Table 5.1 and Table 5.2. All tests were realized on 32 x 32 data set.

The experiment were held on two datasets:

- **Dataset 1** – extended, hard included, cropped & bordered and not simplified data set.
- **Dataset 2** – extended, hard not included, cropped & bordered and not simplified data set.

The best performing model[1] on Dataset 1 has achieved **test error 33.3%**. Result of model best performing on training data of Dataset 2 was estimated to reach **test error 26.8%**.

Data set without hard images highly outperformed the other one. However, the test errors are not good to compare. The size of each data set is different, therefore they were shuffled differently (this holds if any parameter other than type of images is different). Most probably the test subset of Dataset 2 contained "very good" images.

Nevertheless, the test errors are comparable to other models evaluated on the same data set.

Findings

- The models were highly unstable. The validation error was flowing +-10% within different folds chosen as validation subsets.
- As expected radial basis function always outperformed linear kernel function.
- With such a low number of components as 20 were still very good models found. The number of components where the results were notably worse was around 13.
- The best performing hyper-parameters of gamma and C were rather low allowing better generalization. There were too few data examples for higher gammas or Cs and the data were highly linearly not-separable. High values of gamma or C resulted in over-fitting.
- The difference in hyper-parameters of best performing models m_1 on Dataset 1 and m_2 Dataset 2 reflects the different size of

1. The experiments have indices 137 and 138 and can be found at *Bachelor-thesis/Programming/Learning/PCA_SVM/results/default/full_cv/out_{137,138}.txt*

Index	Components	C	Gamma	Kernel f.	Time	Val. error
95	50	12	0.01	rbf	3.77m	31.8%
94	100	8	0.005	rbf	9.16m	33.4%
97	60	12	0.01	rbf	4.07m	34.0%
96	40	12	0.01	rbf	3.69m	34.7%
24	120	12	0.0075	rbf	53.07m	35.5%
66	20	20	0.005	rbf	5.93m	36.0%
0	150	10	0.0075	rbf	–	36.5%
51	50	12	0.005	linear	36.89m	36.8%
78	20	100	0.01	linear	55.80m	42.6%
44	12	450	0.0075	rbf	–	47.0%
45	12	50	0.0075	linear	–	53.4%
102	50	1	1	rbf	7.77m	69.4%
100	50	50	1	rbf	6.86m	69.4%

Table 5.1: PCA & SVM results on Dataset 1. The second column specifies how many components were set to be discovered by PCA.

those data sets. E.g. gamma in m_2 was lower in order to better generalize smaller data.

5.2 Convolutional neural network

Because of its input assumption, CNN was expected to perform with the lowest test error. Two different architectures were designed after first experiments.

Let there be a validation data set X_v and training data set X_t. The learning process was conducted in batches. Ordinary batch consisted of 100 training data examples. The validation frequency constant VF was set, usually being 30. After model was trained on VF batches, the

Index	Components	C	Gamma	Kernel f.	Time	Val. error
132	50	14	0.005	rbf	1.56m	31.8%
105	50	12	0.01	rbf	1.61m	32.9%
38	120	10	0.0075	rbf	28.38m	34.0%
36	20	10	0.0075	rbf	8.83m	37.3%
39	120	10	0.0075	sigmoid	29.06m	43.1%

Table 5.2: PCA & SVM results on Dataset 2.

network was fed up with validation data set and its validation error was recorded.

Train validation condition noted as TVC was determined, usually to 15. Each time VF batches were trained the training-finish check was conducted. If last TVC validation errors contained only one local minimum lm, the training procedure ended. The model used after batch raising validation error lm was proclaimed as the resulting model.

5.2.1 CNN1

The first smaller CNN had following architecture:

$$IN -> CL_1(5x5) -> PL_1 -> CL_2(3x3) -> PL_2 -> FC -> OL \tag{5.1}$$

IN has been usually of shape 32x32x3. First CL_1 has had *stride* (1,1) and filter size 5x5. Following PL_1 has ordinary *receptive field* and *stride* equal to 2x2.

Second CL_2 had size of its filters only 3x3 and depth around twice deeper comparing to CL_1's depth. Again *stride* (1,1) was used. Set up of PL_2 was equal to PL_1.

Fully connected layer FC consisted of various number of neurons, being the suspect of investigation. The last fully-connected output layer contained 7 neurons, predicting the class probabilities.

Cross entropy was used as loss function. Neurons in FC layers were using *sigmoid* activation function. Neurons in CL were using *ramp function*. Activations in OL were normalized into class probabilities using *soft-max function*.

Considered data sets:

- **Dataset 1** – as defined in Section 5.1
- **Dataset 3** – not extended (only 1240 original images), hard included, cropped & bordered and not simplified data set.
- **Dataset 4** – not extended, hard included, cropped & bordered and simplified data set.
- **Dataset 5** – extended, hard included, cropped & bordered and simplified data set.

  The results on those data sets can be seen Tables 5.3, 5.4, 5.5 and 5.6.

  The best model on Dataset 1 performed test error of **29.9%** with index 43, best model on Dataset 3 scored test error **36.2%** with index 48. Best fit for Dataset 4 achieved test error **27.9%** with index 47. Lastly, best performing model on Dataset 5 reached test error **23.1%** with experiment's id 46.

  Findings

  - The models were much more stable comparing to PCA & SVM. Most of the cross-validations were fluctuating around +-2% difference in validation error over different validation folds. Thus, CNN generalizes better.
  - When using lower number of filters in convolutional layers, good results could be still obtained by lowering the dropout hyper-parameter.
  - The difference in performance between extended and not extended data set was noticeable.
  - Small starting learning rate was the key to good predictions. With higher learning rate, the model was not achieving even 40% validation error rate.
  - Much more computational expensive than PCA & SVM.

or i	fc1	m	t	dp	cfs	cfd	csd	ve
40	1300	0.95	87.06m	0.25	1	30	60	24.8%
39	1300	0.95	1.73h	0.4	1	70	140	26.8%
8	1500	0.95	-	0.6	2	85	160	27.0%
24	1300	0.95	-	0.35	1	30	60	27.5%
30	1300	0.97	45.05m	0.25	1	30	60	28.1%
11	1300	0.95	-	0.4	1	20	40	29.3%

Table 5.3: CNN1 results on Dataset 1. The *or i* stand for the index of the test. *fc1* describes how many neurons were used in the hidden fully connected layer, *m* denotes momentum, *t* denotes time, *dp* stands for dropout probability, *cfs* denotes the stride used in the first CL, *cfd* means the number of filters in the first CL, *csd* describes the number of filter in the second CL. Last column denotes the 5-fol cross validation error. Important note is that tests with lower or equal index to 27 were run on slightly different cross-validation and could perform up to 1% better. The size of the filter in first CL was always 5x5 and 3x3 in second's. Staring learning rate was 0.0005. The VF was set to 30. With lower VF obtaining worse results.

- A lot of hyper-parameters make the tuning process very complicated and time expensive. Once good parameters have been found, the CNN performs well on almost any type of data set.

5.2.2 CNN2

$CNN2$ is almost consistent with the $CNN1$, however it is a little bit deeper. It has extra CL and PL before the FC layer.

or i	fc1	m	t	dp	cfs	cfd	csd	ve
32	1300	0.95	71.23m	0.25	1	70	140	31.2%
33	1300	0.95	69.68m	0.4	1	70	140	31.8%
31	1300	0.95	44.65m	0.25	1	30	60	33.6%
0	1300	0.95	-	0.6	2	75	150	34.2%
2	1300	0.95	-	0.6	2	75	150	34.4%

Table 5.4: CNN1: Dataset 3

or i	fc1	m	t	dp	cfs	cfd	csd	ve
34	1300	0.95	59.71m	0.4	1	70	140	24.2%
35	1300	0.95	38.61m	0.25	1	30	60	26.2%

Table 5.5: CNN1: Dataset 4

or i	fc1	m	t	dp	cfs	cfd	csd	ve
36	1300	0.95	53.64m	0.25	1	30	60	18.6%
37	1300	0.95	47.24m	0.4	1	30	60	19.0%
38	1300	0.95	86.41m	0.4	1	70	140	20.1%

Table 5.6: CNN1: Dataset 5

or i	fc1	dp	cfs	cfd	csd	cst	ve
40	1300	0.33	1	30	45	60	22.7%
5	1500	0.5	2	30	70	140	24.4%
21	1300	0.45	2	30	70	140	25.0%
27	1300	0.6	2	33	66	132	25.1%
11	1100	0.5	2	30	70	140	25.7%
7	1100	0.5	2	35	70	140	26.1%

Table 5.7: Validation errors from 5-fold cross validation over training data with *CNN2*. All filter sizes are 3x3. *cst* denotes the depth of the third CL. Parameters such as learning and others are consistent with *CNN1*. Momentum was always 0.95.

Therefore, the filter's sizes are normalized to be 3x3 in all CLs. Strides of first CL is either (1,1) or (2,2). Stride of other CLs is always (1,1). The strides of PLs are (2,2). Other specifics are the same as for $CNN1$.

The $CNN2$ was tested on the main Dataset 1. It was relatively hard to obtain bad results with $CNN2$, however reaching outstanding results was also difficult.

The best model (Figure 5.7) scored the **test error 28.7%** (index number 41) with confusion matrix corresponding to Figure 5.8

5.2.3 Autoencoder

Simple autoencoder was tried as a potential model for classification.

$$IL- > FC- > OL \tag{5.2}$$

Where OL has the same shape as IL and it is expected its activations to be similar to IL as much as possible. The mean squared error loss function was used for fitting the autoencoder. FC uses ReLU unit, OL uses sigmoid function as its activation function. Training data are

	Blue	Box	Can	Chem.	Colo.	Gree.	Whit	Pred
Blue	221	8	2	7	0	0	37	80.4%
Box	12	168	64	23	1	4	3	61.1%
Can	6	6	202	9	27	15	10	73.5%
Chem.	26	63	20	102	0	22	42	37.1%
Colo.	0	6	7	6	234	0	22	85.1%
Green	11	0	0	1	0	262	1	95.3%
White	42	5	24	12	8	0	184	66.9%

Table 5.8: *CNN2*: Confusion matrix of run with index 41 and test error 28.7%. Columns are predicted values, rows are actual.

split into validation and training data set, they are used for fitting the autoencoder (only for one fold, not cross-validation).

After fitting the autoencoder, for classification, the encoder part was used:

$$IL- > FC \tag{5.3}$$

Each image was fed into the IL and the activations in FC were recorded as *codings*. The *codings* were then used as the inputs for feed-forward neural network consisting of IL, one FC hidden layer and fully-connected OL. The number of neurons in OL were consistent with number of classes: 7. Usual 5-fold cross-validation was used for measuring performance as described in Section 5.

The best model scored on Dataset 1 the **test error 36.8%**

5.2.4 Deep Autoencoder

As for the deep autoencoder the following structure was tested:

$$IL- > 128- > 64- > 32- > 64- > 128- > OL \tag{5.4}$$

The number stands for the number of neurons in the hidden layer. The result of encoding and decoding using the trained deep autoencoder can be seen in Figure 5.2. The result achieved when classifying the

Index	Epochs	Fc1	Val Err
13	750	170	33.3%
10	750	160	35.0%
9	750	150	35.9%
8	1000	150	36.7%
7	1000	140	37.2%
4	1000	200	37.6%
2	1000	180	38.0%
6	1000	170	38.1%

Table 5.9: Autoencoder: Results of predictions on autoencoder's encodings of Dataset 1 images. *Epochs* is the number for epochs spent with training. *Fc1* denotes the number of neurons in hidden layer. The neural networks (NNs) executions with indices higher than 7 are using the most precise autoencoder with the size of encoding layer 128 neurons, adadelta optimization algorithm and 2000 epochs of training the autoencoder. The ones with lower index use only 32 neurons in encoding layer.

images based on theirs encodings was not favorable. The test accuracy on Dataset 1 was only 54%.

On the result can be looked on as the proof that better compressing abilities using deep autoencoder does not have to be good for classification. Thus, CNNs should be best for image classification instead.

Furthermore, only CNNs were researched within this thesis.

5.2.5 Other models

As part of the study, other models and approaches were tested, including usage of edge detectors, Residual networks or other resolutions of images.

Canny edge detection algorithm was used to find edges on the image. The edges were than used in PCA & SVM for classification on simplified data set, however without satisfactory results.

1. Corresponding log file with index 0

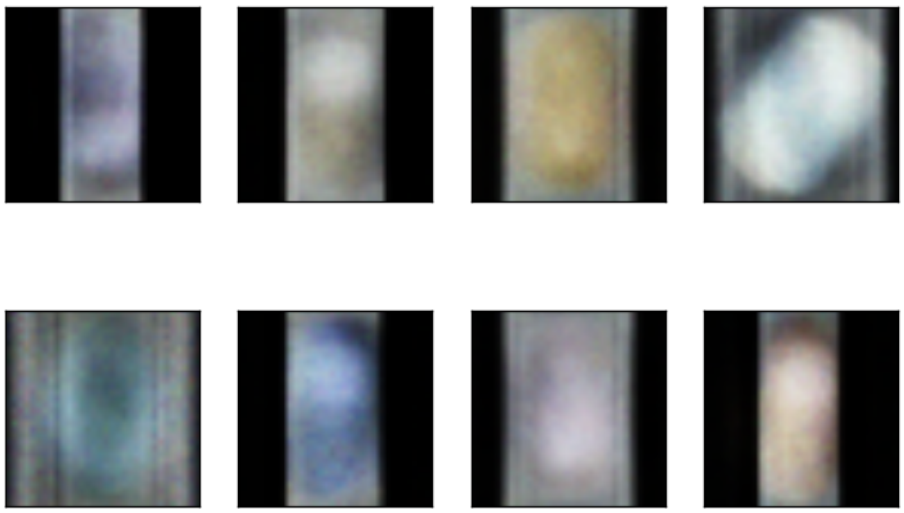

Figure 5.2: 8 images encoded and decoded using deep autoencoder network with number of neurons in hidden layers: 128,64,32,64,128.

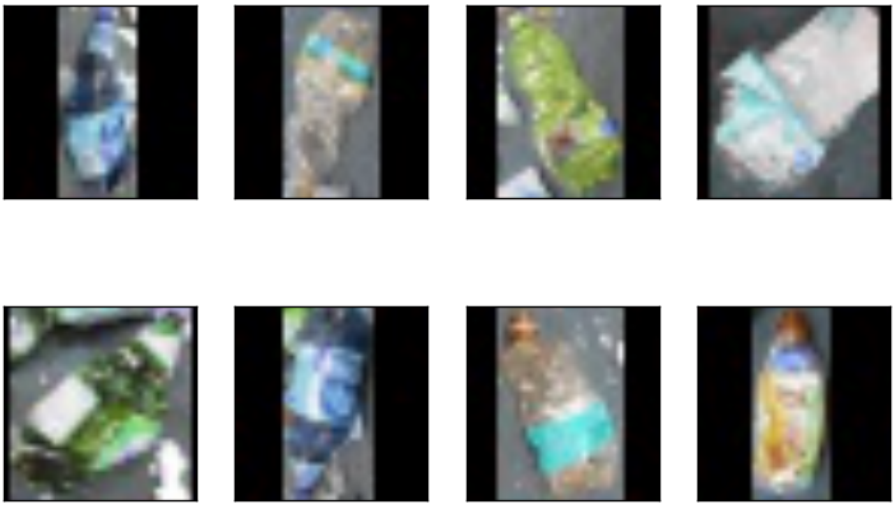

Figure 5.3: Originals of images encoded and decoded in Figure 5.2

Image with edges were used to feed CNN as well to classify simplified data set. The top result conducted on Dataset 5 with resolution 48x48 was 5-fold cross validation error 38.7%[2]. Considering there were only three classes the result was unsatisfying.

Further image of edges was added as a fourth channel to the input of CNN. The results were worse by 8% comparing to three channels version. It proves the theorem that CNN's filters discover edges, better. One good result with 4 channels approach was only on Dataset 5 with resolution 32x32 ending at 5-fold cross validation error 22.8%[3].

Residual networks are deep convolutional networks without any regularization techniques used, without almost any pooling layers, many CLs stacked together forming blocks. Some layers do not receive input only from the previous layer, but also other layer further behind [6].

When testing Residual networks within *Bachelor-thesis* project, the over-fitting was a decent problem, which have not been solved within the thesis. Many architectures were tested. The ones with lower number of parameters provided better results, however always, after some milestone the model started to over-fit. The best validation accuracy on random split of data was around 34%.

Bigger resolution has come with a trade-of: either bigger *strides* must be put or the computation will be easily over-fitting and computationally expensive. A few experiments were held on 48x48 data set, however the focus stayed on 32x32 resolution, since there was still a space for improvement.

5.2.6 Real environment simulation

Real environment situation was conducted on the data set with original scaled images.

Since we do not know how well would potential mechanic separator/handle separate the object from the rest in order to take a picture under the optical lens and classify them. Therefore, there were version of real environment simulation held including the hard images. Hard images could contain multiple objects as shown in Figure 5.4. The

2. Its index is 9 and can be found at */Learning/CNN/results/edges/full_cv/out_9.txt*
3. */Learning/CNN/results/4_channels/full_cv/out_2.txt*

Figure 5.4: Colorful bottle (64x48) at top-left, box at bottom-left, green bottle at top-right and cans at bottom-right. All images consists of multiple various objects. It can be seen the pictures were even taken from various angles and distances.

objects could be relatively small as well. The resolution used for test was 32x24.

The architecture and type of model used for the simulation was consistent with the $CNN1$ model and the final score reached **text error 35.2%** what is relatively good considering how much harder it can be for CNN to classify the original images (see Figure 5.4).

6 Conclusion

This thesis compared various approaches in order to find automatic waste sorting algorithm. The PCA & SVM resulted in computationally very fast method with little and easy to tune up hyper-parameters. The negative of PCA & SVM method was its instability resulting in the highest difference between test and validation error.

The next method evaluated within this thesis was autoencoder, the neural network trying to compress the image and reconstruct the original from compressed vector. Even though autoencoder is good for compression tasks, it is not widely used for classification, what was confirmed in this thesis. However, autoencoder proved to be better solution than PCA & SVM and has had the smallest difference between test and validation error.

The best method for classification of waste was Convolutional neural networks. *CNN2* scoring test error on Cropped data as little as 28.7% on the major data set Dataaset 1, resulting in test error within real environment 35%. Stacking more CLs proved to raise better results.

However, the results are not satisfactory, rather only leads. The accuracy over 80% was achieved with *CNN1* only when restricting the data set only into three classes (what can be useful for the waste sorting company).

The main confusion brings Chemical category as seen on Table 5.8. Furthermore, despite Box having a special shape, it is frequently mislead with Can. On the other side, Green category shows almost perfect results[1].

The future work in this matter must be followed. Creating larger data set, exploring the right set up of Residual networks, those are the patters where the research should continue. This thesis should be a good introduction into the the problem and its limits.

1. It is easily recognizable color – the reason why is so favored as a background of movie tricks.

Bibliography

[1] Ian Goodfellow, Yoshua Benigo, and Aaron Courville. *Deep Learning*. The MIT Press, 2016. 800 pp. ISBN: 0-262-03561-8.

[2] Geoffrey E. Hinton and Simon Osindero. "A fast learning algorithm for deep belief nets". In: (). URL: https://www.cs.toronto.edu/~hinton/absps/fastnc.pdf.

[3] Kevin P. Murphy. *Machine Learning. A Probabilistic Perspective*. The MIT Press, 2012. 1104 pp. ISBN: 0-262-01802-0.

[4] Yoshua Bengio. "Learning Deep Architectures for AI". In: (). URL: http://www.iro.umontreal.ca/~lisa/pointeurs/TR1312.pdf (visited on 12/15/2016).

[5] *Convolutional Neural Networks (CNNs / ConvNets)*. 2016. URL: http://cs231n.github.io/convolutional-networks/ (visited on 09/01/2017).

[6] Kaiming He et al. "Deep Residual Learning for Image Recognition". In: (). URL: https://arxiv.org/abs/1512.03385 (visited on 12/15/2016).

[7] G. E. Hinton and R. R. Salakhutdinov. "Reducing the Dimensionality of Data with Neural Networks". In: (). URL: http://science.sciencemag.org/content/313/5786/504.

[8] I.T. Jolliffe. *Principal Component Analysis*. Springer, 2002. 487 pp. ISBN: 978-0-387-95442-4.

[9] Nello Cristianini and John Shawe-Taylor. *An Introduction to Support Vector Machines and Other Kernel-based Learning Methods*. Cambridge University Press, 2000. ISBN: 0-521-78019-5.

A Appendix – list of attachments

Source code of *Bachelor-thesis* together with all images and results of experiments can be found at: `https://github.com/jodik/Bachelor-thesis`.

Together with the text of the thesis, the *Bachelor-thesis* project with its data is attached. Further attachments consists of further results which could not be contained in the thesis and steps of dependencies to run the project.

B Appendix – project configuration

For running the project following must be installed:

1. sudo apt-get install python-pip python-dev
2. sudo pip install tensorflow
3. export PYTHONPATH="$PYTHONPATH:/path/to/thesis/Bachelor-thesis"
4. sudo pip install texttable
5. sudo pip install keras
6. sudo pip install sklearn
7. sudo pip install cobs
8. sudo pip install enum
9. sudo pip install matplotlib
10. apt-get install python-tk
11. sudo apt-get install python-opencv

Tested on *Debian 3.16.36-1deb8u2 (2016-10-19) x86_64 GNU/Linux*

C Appendix – full results

Index	Components	C	Gamma	Kernel f.	Time	Val. error
95	50	12	0.01	rbf	3.77m	31.8%
48	50	12	0.01	rbf	3.92m	31.8%
63	50	12	0.01	rbf	5.39m	31.8%
104	50	10	0.01	rbf	2.96m	31.9%
49	50	12	0.0075	rbf	4.36m	32.0%
55	50	12	0.0075	rbf	5.27m	32.0%
57	50	50	0.0075	rbf	5.01m	32.1%
54	50	8	0.0075	rbf	5.31m	32.1%
56	50	20	0.0075	rbf	5.16m	32.2%
61	50	50	0.005	rbf	5.45m	32.4%
60	50	100	0.005	rbf	5.20m	32.7%
42	50	10	0.0075	rbf	3.16m	32.9%
53	50	10	0.0075	rbf	5.00m	32.9%
47	50	12	0.005	rbf	3.83m	33.2%
64	50	12	0.005	rbf	5.80m	33.2%
94	100	8	0.005	rbf	9.16m	33.4%
93	100	12	0.005	rbf	10.67m	33.7%
62	50	50	0.01	rbf	5.12m	33.8%
58	50	100	0.0075	rbf	4.95m	34.0%
97	60	12	0.01	rbf	4.07m	34.0%
84	100	8	0.0075	rbf	13.40m	34.1%

Table C.1: PCA & SVM: Dataset 1

Index	Components	C	Gamma	Kernel f.	Time	Val. error
59	50	100	0.01	rbf	5.13m	34.1%
92	100	20	0.005	rbf	10.57m	34.6%
91	100	20	0.005	rbf	11.57m	34.6%
96	40	12	0.01	rbf	3.69m	34.7%
85	100	12	0.0075	rbf	12.88m	35.1%
83	100	8	0.01	rbf	13.87m	35.2%
82	100	12	0.01	rbf	13.73m	35.5%
24	120	12	0.0075	rbf	53.07m	35.5%
5	120	12	0.0075	rbf	7.64m	35.5%
81	100	20	0.01	rbf	13.65m	35.7%
89	100	50	0.005	rbf	11.28m	36.0%
90	100	50	0.005	rbf	11.40m	36.0%
66	20	20	0.005	rbf	5.93m	36.0%
72	20	12	0.0075	rbf	7.63m	36.1%
79	100	100	0.01	rbf	13.46m	36.2%
41	20	10	0.0075	rbf	2.74m	36.2%
65	20	12	0.005	rbf	5.47m	36.2%
73	20	12	0.01	rbf	7.79m	36.3%
0	150	10	0.0075	rbf	unknown	36.5%
88	100	100	0.0075	rbf	11.95m	36.8%
25	130	12	0.0075	rbf	57.67m	36.8%
51	50	12	0.005	linear	36.89m	36.8%
50	50	12	0.0075	linear	37.08m	36.8%
52	50	12	0.01	linear	37.08m	36.8%
76	20	100	0.01	rbf	8.79m	36.9%

Table C.2: PCA & SVM: Dataset 1

87	100	50	0.0075	rbf	12.46m	37.0%
68	20	100	0.005	rbf	6.63m	37.1%
Index	Components	C	Gamma	Kernel f.	Time	Val. error
67	20	50	0.005	rbf	6.12m	37.2%
74	20	20	0.01	rbf	7.87m	37.5%
43	20	50	0.0075	rbf	2.86m	37.5%
70	20	50	0.0075	rbf	6.94m	37.5%
71	20	50	0.0075	rbf	7.72m	37.5%
6	20	12	0.0075	rbf	4.44m	37.9%
69	20	100	0.0075	rbf	6.38m	38.3%
7	20	50	0.0075	rbf	12.29m	38.9%
3	150	10	0.0075	rbf	unknown	39.4%
10	20	120	0.0075	rbf	13.52m	39.4%
78	20	100	0.01	linear	55.80m	42.6%
99	50	50	0.1	rbf	6.19m	45.3%
98	50	12	0.1	rbf	6.14m	45.6%
44	12	450	0.0075	rbf	13.40h	47.0%
103	50	1	0.1	rbf	6.41m	47.5%
77	20	100	0.01	sigmoid	9.06m	51.6%
45	12	50	0.0075	linear	13.64h	53.4%
102	50	1	1	rbf	7.77m	69.4%
101	50	12	1	rbf	6.78m	69.4%
100	50	50	1	rbf	6.86m	69.4%

Table C.3: PCA & SVM: Dataset 1

Index	Components	C	Gamma	Kernel f.	Time	Val. error
132	50	14	0.005	rbf	1.56m	31.8%
120	50	15	0.005	rbf	2.74m	31.9%
131	50	10	0.006	rbf	1.55m	32.0%
129	50	12	0.006	rbf	1.57m	32.0%
107	50	12	0.005	rbf	1.68m	32.3%
134	50	13	0.005	rbf	86.78s	32.3%
121	50	20	0.005	rbf	2.73m	32.3%
119	50	10	0.005	rbf	2.57m	32.5%
124	50	12	0.002	rbf	2.86m	32.5%
106	50	12	0.0075	rbf	1.64m	32.7%
34	50	12	0.0075	rbf	14.80m	32.7%
130	50	15	0.006	rbf	84.94s	32.7%
133	50	16	0.005	rbf	88.50s	32.7%
37	50	10	0.0075	rbf	14.48m	32.8%
105	50	12	0.01	rbf	1.61m	32.9%
118	50	8	0.005	rbf	2.55m	32.9%
128	50	12	0.004	rbf	1.61m	33.1%
127	50	5	0.002	rbf	2.57m	33.2%
123	50	15	0.002	rbf	2.87m	33.3%
122	50	20	0.002	rbf	2.81m	33.4%
125	50	10	0.002	rbf	2.88m	33.6%

Table C.4: PCA & SVM: Dataset 2

Index	Components	C	Gamma	Kernel f.	Time	Val. error
126	50	8	0.002	rbf	2.75m	33.6%
38	120	10	0.0075	rbf	28.38m	34.0%
33	120	12	0.0075	rbf	28.84m	34.4%
30	120	12	0.0075	rbf	31.19m	34.4%
117	120	5	0.005	rbf	3.51m	35.9%
115	120	20	0.01	rbf	2.71m	36.1%
116	120	20	0.0075	rbf	2.45m	36.4%
114	120	50	0.01	rbf	2.65m	37.1%
36	20	10	0.0075	rbf	8.83m	37.3%
35	20	12	0.0075	rbf	8.67m	37.6%
110	20	20	0.01	rbf	1.65m	38.6%
111	20	20	0.0075	rbf	1.62m	38.8%
108	20	12	0.01	rbf	1.58m	38.9%
109	20	12	0.01	rbf	1.58m	38.9%
113	20	50	0.01	rbf	1.60m	40.6%
112	20	50	0.0075	rbf	1.59m	40.9%
40	20	10	0.0075	sigmoid	8.57m	41.9%
39	120	10	0.0075	sigmoid	29.06m	43.1%

Table C.5: PCA & SVM: Dataset 2

or i	fc1	m	t	dp	cfs	cfd	csd	ve
40	1300	0.95	87.06m	0.25	1	30	60	24.8%
28	1300	0.95	48.42m	0.22	1	30	60	26.0%
29	1300	0.95	39.92m	0.27	1	30	60	26.4%
39	1300	0.95	1.73h	0.4	1	70	140	26.8%
20	1300	0.95	-	0.425	1	70	140	26.8%
8	1500	0.95	-	0.6	2	85	160	27.0%
26	1300	0.95	-	0.30	1	30	60	27.5%
24	1300	0.95	-	0.35	1	30	60	27.5%
19	1300	0.95	-	0.375	1	70	140	27.5%
7	1300	0.95	-	0.6	2	75	150	27.5%
12	1300	0.95	-	0.4	1	30	60	27.7%
14	1300	0.95	-	0.4	1	50	100	27.7%
9	1300	0.95	-	0.6	1	75	150	27.7%
27	1300	0.95	-	0.2	1	30	60	27.9%
22	1300	0.95	-	0.25	1	30	60	27.9%
21	1300	0.95	-	0.45	1	70	140	27.9%
30	1300	0.97	45.05m	0.25	1	30	60	28.1%
10	1300	0.95	-	0.4	1	75	150	28.3%
13	1300	0.95	-	0.4	1	40	80	28.5%
5	1300	0.95	-	0.6	2	75	150	28.5%
4	1300	0.95	-	0.6	2	75	150	28.6%

Table C.6: CNN1: Dataset 1

or i	fc1	m	t	dp	cfs	cfd	csd	ve
25	1300	0.95	-	0.35	1	30	60	28.8%
15	1300	0.95	-	0.4	1	60	120	28.9%
17	1300	0.95	-	0.4	1	80	190	28.9%
18	1300	0.95	-	0.4	1	90	180	29.1%
11	1300	0.95	-	0.4	1	20	40	29.3%
6	1300	0.95	-	0.6	2	75	150	33.9%

Table C.7: CNN1: Dataset 1

or i	fc1	m	t	dp	cfs	cfd	csd	ve
32	1300	0.95	71.23m	0.25	1	70	140	31.2%
33	1300	0.95	69.68m	0.4	1	70	140	31.8%
31	1300	0.95	44.65m	0.25	1	30	60	33.6%
0	1300	0.95	-	0.6	2	75	150	34.2%
2	1300	0.95	-	0.6	2	75	150	34.4%

Table C.8: CNN1: Dataset 3

or i	fc1	m	t	dp	cfs	cfd	csd	ve
34	1300	0.95	59.71m	0.4	1	70	140	24.2%
35	1300	0.95	38.61m	0.25	1	30	60	26.2%

Table C.9: CNN1: Dataset 4

or i	fc1	m	t	dp	cfs	cfd	csd	ve
36	1300	0.95	53.64m	0.25	1	30	60	18.6%
37	1300	0.95	47.24m	0.4	1	30	60	19.0%
38	1300	0.95	86.41m	0.4	1	70	140	20.1%

Table C.10: CNN1: Dataset 5

or i	fc1	dp	cfs	cfd	csd	cst	ve
40	1300	0.33	1	30	45	60	22.7%
19	1500	0.6	2	30	70	140	24.3%
5	1500	0.5	2	30	70	140	24.4%
10	1100	0.5	2	30	60	125	24.7%
4	1300	0.5	2	30	70	140	24.8%
20	1500	0.45	2	30	70	140	24.8%
21	1300	0.45	2	30	70	140	25.0%
8	1100	0.5	2	27	70	140	25.1%
27	1300	0.6	2	33	66	132	25.1%
15	1500	0.5	2	30	60	130	25.1%
9	1100	0.5	2	30	60	140	25.5%
29	1300	0.6	2	39	78	156	25.5%
32	1700	0.6	2	33	66	132	25.6%
30	1700	0.6	2	39	78	156	25.6%
6	1100	0.5	2	30	70	140	25.7%
11	1100	0.5	2	30	70	140	25.7%
13	1500	0.5	2	27	70	140	25.8%
2	1300	0.5	2	50	80	160	25.9%
22	1300	0.6	2	30	70	140	25.9%
28	1300	0.6	2	36	72	144	25.9%

Table C.11: CNN2: Dataset 1

or i	fc1	dp	cfs	cfd	csd	cst	ve
16	1100	0.45	2	30	60	125	26.0%
17	1100	0.45	2	30	60	125	26.0%
7	1100	0.5	2	35	70	140	26.1%
34	1700	0.55	2	30	70	140	26.2%
33	1700	0.6	2	30	70	1402	26.2%
35	1700	0.65	2	30	70	140	26.2%
14	1500	0.5	2	30	60	140	26.5%
31	1500	0.6	2	33	66	132	26.5%
0	1300	0.5	2	40	70	140	26.6%
18	1100	0.6	2	30	60	125	26.9%
25	1300	0.6	2	24	48	96	27.1%
3	1300	0.5	1	15	70	140	27.2%
26	1300	0.6	2	27	54	108	27.2%
24	1300	0.6	2	21	42	84	27.3%
12	1500	0.5	2	35	70	140	27.7%
1	1300	0.5	2	40	150	70	28.2%
23	1300	0.6	2	18	36	72	29.3%
52	1300	0.33	1	30	45	60	30.3%

Table C.12: CNN2: Dataset 1

Or. i.	Epochs	Fc1	Val Err
13	750	170	33.3%
10	750	160	35.0%
9	750	150	35.9%
8	1000	150	36.7%
7	1000	140	37.2%
5	1000	150	37.3%
4	1000	200	37.6%
3	1000	160	37.8%
2	1000	180	38.0%
6	1000	170	38.1%
0	500	160	40.7%
1	500	160	41.1%

Table C.13: Autoencoder: Dataset 1. The neural networks (NNs) executions with indices higher than 7 are using the most precise autoencoder with the size of encoding layer 128 neurons, adadelta optimization algorithm and 2000 epochs of training the autoencoder. The ones with lower index use only 32 neurons in encoding layer.