COMENIUS UNIVERSITY IN BRATISLAVA
FACULTY OF MATHEMATICS, PHYSICS AND INFORMATICS

HANDWRITTEN CHARACTER RECOGNITION USING MACHINE LEARNING METHODS

Bachelor's Thesis

Study Program:	Applied Informatics
Branch of Study:	2511 Applied Informatics
Educational Institution:	Department of Applied Informatics
Supervisor:	Mgr. Ľudovít Malinovský

Bratislava, 2013

Ivor Uhliarik

Comenius University in Bratislava
Faculty of Mathematics, Physics and Informatics

THESIS ASSIGNMENT

Name and Surname:	Ivor Uhliarik
Study programme:	Applied Computer Science (Single degree study, bachelor I. deg., full time form)
Field of Study:	9.2.9. Applied Informatics
Type of Thesis:	Bachelor´s thesis
Language of Thesis:	English
Secondary language:	Slovak

Title: Handwritten character recognition using machine learning methods

Aim: The student's task is to review existing machine learning based handwritten character recognition methods and implement one of them for an Adnroid application. The resulting application will be able to:
* recognize characters written by the user
* recognize characters from camera or a picture
* learn new characters
* learn interactively based on user's feedback
The student also reviews about the state of the art in similar applications.

Annotation: Thesis briefly reviews machine learning methods for handwritten character recognition and smartphone applications which employ them for similar purpose. In addition, a similar user friendly application is developed.

Supervisor:	Mgr. Ľudovít Malinovský
Department:	FMFI.KAI - Department of Applied Informatics
Vedúci katedry:	doc. PhDr. Ján Rybár, PhD.
Assigned:	19.10.2012
Approved:	29.10.2012

doc. RNDr. Mária Markošová, PhD.
Guarantor of Study Programme

Uhliarik
.....
Student

.....
Supervisor

Súhlasím s tým, že práca bude vypracovaná v angličtine.

Acknowledgment

I would like to express my sincere gratitude to my supervisor Mgr. Ľudovít Malinovský for invaluable consultations, interest, initiative, and continuous support throughout the duration of writing this thesis. I would also like to thank my family, fellow colleagues and all the people that have supported me. Finally, I would like to thank Martin Boze and the rest of the n'c community for listening to my rants and all the witty remarks.

Declaration on Word of Honor

I declare that this thesis has been written by myself using only the listed references and consultations provided by my supervisor.

Bratislava, …..... 2013

….....

Ivor Uhliarik

Abstract

The aim of this work is to review existing methods for the handwritten character recognition problem using machine learning algorithms and implement one of them for a user-friendly Android application. The main tasks the application provides a solution for are handwriting recognition based on touch input, handwriting recognition from live camera frames or a picture file, learning new characters, and learning interactively based on user's feedback. The recognition model we have chosen is a multilayer perceptron, a feedforward artificial neural network, especially because of its high performance on non-linearly separable problems. It has also proved powerful in OCR and ICR systems [1] that could be seen as a further extension of this work. We had evaluated the perceptron's performance and configured its parameters in the GNU Octave programming language, after which we implemented the Android application using the same perceptron architecture, learning parameters and optimization algorithms. The application was then tested on a training set consisting of digits with the ability to learn alphabetical or different characters.

Keywords: Character recognition, Multilayer Perceptron, Backpropagation, Rprop, Image processing

Abstrakt

Cieľom tejto práce je ponúknuť prehľad metód rozpoznávania rukou písaných znakov pomocou algoritmov strojového učenia a jeden z nich implementovať vo forme Android aplikácie priateľskej pre používateľov. Hlavnými požiadavkami na túto aplikáciu sú rozpoznávanie rukou písaného písma pomocou dotykových senzorov Android zariadenia a zo záberov kamery alebo súboru s obrázkom, učenie nových, dosiaľ nepoznaných znakov a interaktívne učenie podľa spätnej väzby používateľa. Ako model učenia sme zvolili doprednú neurónovú sieť – viacvrstvový perceptrón. Voľbu ovplyvnil vysoký výkon na nelineárne separovateľných problémoch, ako aj fakt, že sa často používa v OCR a ICR systémoch [1], ktoré by sa dali prirovnať rozšíreniam tejto práce. Pre testovanie a vyhodnotenie výkonu, ako aj správne nastavenie parametrov perceptrónu sme použili programovací jazyk GNU Octave. Na základe dosiahnutej konfigurácie sme ho implementovali vo vytvorenej Android aplikácii, pričom sme použili rovnakú architektúru, parametre a algoritmy optimalizácie. Aplikáciu sme testovali na trénovacej sade cifier s možnosťou pridania a naučenia alfabetických a iných znakov.

Kľúčové slová: Rozpoznávanie znakov, Viacvrstvový perceptrón, Backpropagation, Rprop, Spracovanie obrazu

Table of Contents

List of Abbreviations and Symbols

OCR	Optical Character Recognition.
ICR	Intelligent Character Recognition.
X	Matrix of all training examples. Each row contains a feature vector of a single example.
x	Feature vector of a training example
y	A vector of labels of training examples.
L	Number of layers in a network.
m	Number of training examples.
Θ	Weights of a neural network. Superscript denotes a layer, subscripts denote index of an element.
$a^{(l)}$	Vector of neuron activations in layer *l*.
h_Θ	Hypothesis of a classifier with given weights Θ.
s_j	The size of layer *j*.
g(z)	Sigmoid function of *z*.
$J(\Theta)$	Cost function of weights Θ.
$\delta^{(l)}$	Vector of error terms for neurons in layer *l*.
$\Delta^{(l)}$	Accumulator of error terms for neurons in layer *l* in the context of pure backpropagation algorithm.
$\Delta^{(t)}$	Matrix of weight update sizes for iteration *t* in the context of RPROP.
$\Delta\Theta^{(t)}$	Matrix of weight change values in the context of RPROP.
Δ_0	Initial weight step size in RPROP.
η^+	Acceleration of weight step size in RPROP.
η^-	Deceleration of weight step size in RPROP.
λ	Network regularization parameter.

Introduction

Handwritten character recognition is a field of research in artificial intelligence, computer vision, and pattern recognition. A computer performing handwriting recognition is said to be able to acquire and detect characters in paper documents, pictures, touch-screen devices and other sources and convert them into machine-encoded form. Its application is found in *optical character recognition* and more advanced *intelligent character recognition* systems. Most of these systems nowadays implement machine learning mechanisms such as *neural networks*.

Machine learning is a branch of artificial intelligence inspired by psychology and biology that deals with learning from a set of data and can be applied to solve wide spectrum of problems. A supervised machine learning model is given instances of data specific to a problem domain and an answer that solves the problem for each instance. When learning is complete, the model is able not only to provide answers to the data it has learned on, but also to yet unseen data with high precision.

Neural networks are learning models used in machine learning. Their aim is to simulate the learning process that occurs in an animal or human neural system. Being one of the most powerful learning models, they are useful in automation of tasks where the decision of a human being takes too long, or is imprecise. A neural network can be very fast at delivering results and may detect connections between seen instances of data that human cannot see.

We have decided to implement a neural network in an Android application that recognizes characters written on the device's touch screen by hand and extracted from camera and images provided by the device. Having acquired the knowledge that is explained in this text, the neural network has been implemented on a low level without using libraries that already facilitate the process. By doing this, we evaluate the performance of neural networks in the given problem and provide source code for the network that can be used to solve many different classification problems. The resulting system is a subset of a complex OCR or ICR system; these are seen as possible future extensions of this work.

In the first chapter, we describe the overall project in greater detail and review existing

approaches, algorithms and systems of similar nature. For overview, we also briefly explain the specific algorithms that have been used in the implementation of the project.

In the second chapter, we explain the chosen neural network model and algorithms on a low level. The information contained in this chapter should be sufficient in order to begin implementing the learning model.

Chapter 3 discusses the design choices made before implementing the Android application. The requirements of the application are specified and the plan of the solution is laid out.

The fourth chapter, implementation, gives a description of how the requirements have been satisfied, what problems have arisen and how they were solved, and also serve as a technical guide for the application users. The structure of the source code is also depicted.

Chapter 5 compares and discusses the numerical results of the network learning algorithms.

Finally, in the conclusion, we talk about the accomplishments of this work and state how the application can be used in other projects or extended to a more complex system.

1. Overview

In order to reach the analysis of the used learning model and the specification and implementation of the algorithms, and consequently, the Android application, we had to review existing approaches to the problem of character recognition. In this chapter, we describe the project and compare the methods that have been considered as candidates for this work.

1.1 Project Description

It has already been stated that the primary goal of our project—an Android application—is able to recognize handwritten characters based on user's touch input and image/camera input in an offline manner. Also, the application provides means of adding and learning a new character and learning interactively from user's feedback.

Android has been selected as the most feasible platform, because many Android devices feature capacitive touch screens, which are very friendly to finger touch input, and cameras, which are necessary to satisfy the camera recognition requirement. Although developing a high quality Android application according to Android development guidelines has not been our primary focus, these have nonetheless not been ignored.

Feature-wise, the application distinguishes two cases: learning from touch input, taking into account both the character bitmap and the individual strokes the user writes, as opposed to learning from a picture bitmap, where we only recognize the character bitmap. Formally, both cases fall into the *offline* approach to handwriting recognition [2]. In this approach, the complete character image is the only information available. *Online recognition*, on the other hand, describes storing the two-dimensional coordinates of points of the writing as a function of time, hence individual strokes are ordered. Therefore, when recognizing user touch input, the use of stroke analysis might resemble the online approach as well because of the additional data provided, even though the strokes are not ordered in time. As such, the system is able to precisely separate certain characters of high bitmap similarity, as in the case of the '8' and the 'B' characters, while being invariant to the order of the strokes the user writes.

We understand interactive learning with user feedback as using *online machine learning*. This should not be confused with online and offline handwriting recognition, which is described above. Online learning is defined as learning one instance of data at a time, expecting feedback—the real label of a character input by a user—to be provided after the neural network's prediction. On the other hand, offline learning performs all of the learning process prior to making any predictions and does not change its prediction hypothesis afterward. The two methods are examples of "lazy" and "eager" learning, respectively. Online machine learning makes the system more adaptive to change, such as a change in trends, prices, market needs, etc. In our case, user feedback makes the system able to adapt to a change in handwriting style, perhaps caused by a change of user. We use both offline and online learning in this work. Initially, the user expects from the application to "just work", so offline learning on initial data is performed. When the system mislabels a character or the user wants to add a new one, they make a correction, refining the system progressively. Moreover, online learning executes after correct prediction as well, so the system should be able to perform better "the more it is used".

In addition to implementing the Android application, we have tested and evaluated the used algorithms in the GNU Octave programming language. The language and its features are dominant in prototyping and make it very easy to visualize data. It is also well suited for machine learning applications as matrix manipulations are part of the language's grammar.

1.2 Character Recognition Algorithms

The algorithms used in character recognition may be divided into three categories: image preprocessing, feature extraction, and classification. They are normally used in sequence—image preprocessing helps make feature extraction a smoother process, while feature extraction is necessary for correct classification.

1.2.1 Image Preprocessing

Image preprocessing is crucial in the recognition pipeline for correct character prediction. These methods typically include noise removal, image segmentation, cropping, scaling, and more. In our project, these methods have mainly been used when recognizing from an

image, but some of them, such as cropping the written character and scaling it to our input size, are also performed in the touch mode.

Digital capture and conversion of an image often introduces noise which makes it hard to decide what is actually a part of the object of interest and what is not. Considering the problem of character recognition, we want to reduce as much noise as possible, while preserving the strokes of the characters, since they are important for correct classification. There are many ways of achieving this. Local processing is one of them.

According to [10], local preprocessing makes use of a small area around the pixel of question in the input image—a mask—to compute the output value of the pixel in the output image. This is called *filtering*. For this task we use convolutional masks that scan an image, ideally reducing all unwanted noise. Masks are square matrices with elements representing weights of the surrounding area pixels that determine the light intensity value of the pixel at hand.

$$h = \frac{1}{9}\begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix} \tag{1.1}$$

Typical average value

$$h = \frac{1}{10}\begin{bmatrix} 1 & 1 & 1 \\ 1 & 2 & 1 \\ 1 & 1 & 1 \end{bmatrix} \tag{1.2}$$

Centroid highlight

In order to preserve character strokes in our project, we have used median filtering, which is a non-linear operation. A median filter replaces the pixel's value with the median of the intensity of surrounding pixels. The result is shown in Figure 1.

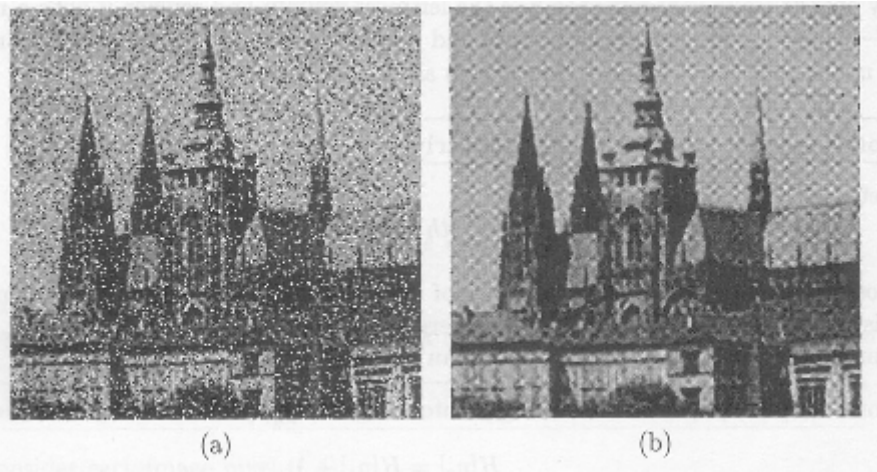


Figure 1: Median filtering: (a) image corrupted with noise; (b) 3x3 median filtering applied to the image [10].

The task of image segmentation is to split an image into parts with strong correlation with objects or the real world properties the image represents [10]. Probably the simplest image segmentation method is *thresholding*. Thresholding is the extraction of the foreground, which is a character in our case, from the rather monotonic background [3]. In a gray-scale image, this is equal to binarization of its real-valued data. An image threshold is selected, according to which the input image is converted. Given an input pixel $f(i, j)$ and threshold T, the output pixel $g(i, j)$ is calculated as follows:

$$g(i,j)=\begin{cases}1 & ,if\ f(i,j)\geq T\\ 0 & ,if\ f(i,j)<T\end{cases} \tag{1.3}$$

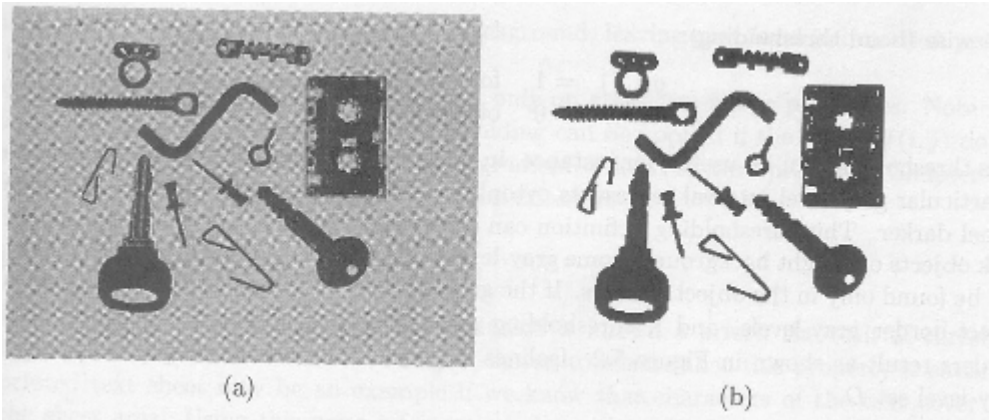


Figure 2: Image thresholding: (a) original image; (b) thresholded segmentation applied to the original image [10].

Finally, in both touch and image based recognition in our work, we have used cropping and scaling of the images to a small fixed size.

1.2.2 Feature Extraction

Features of input data are the measurable properties of observations, which one uses to analyze or classify these instances of data. The task of feature extraction is to choose relevant features that discriminate the instances well and are independent of each other.

According to [3], selection of a feature extraction method is probably the single most important factor in achieving high recognition performance. There is a vast amount of methods for feature extraction from character images, each having different characteristics, invariance properties, and reconstructability of characters. [3] states that in order to answer to the question of which method is best suited for a given situation, an experimental evaluation must be performed.

The methods explained in [3] are template matching, deformable templates, unitary image transforms, graph description, projection histograms, contour profiles, zoning, geometric moment invariants, Zernike moments, spline curve approximation, and Fourier descriptors. To describe the way feature extraction is sometimes done in handwriting recognition, we briefly explain one of them.

Projection histograms were introduced in 1956 in an OCR system by Glauberman [6] and are used in segmentation of characters, words, and text lines, or to detect if a scanned text page is rotated [3]. We collect the horizontal and vertical projections of an image by setting each horizontal and vertical “bin” value to the count of pixels in respective rows and columns (Figure 3), where neighboring bins can be merged to make the features scale independent. The projection is, however, variant to rotation and variability in writing style. The two histograms are then compared and a dissimilarity measure is obtained, which can be used as a feature vector.

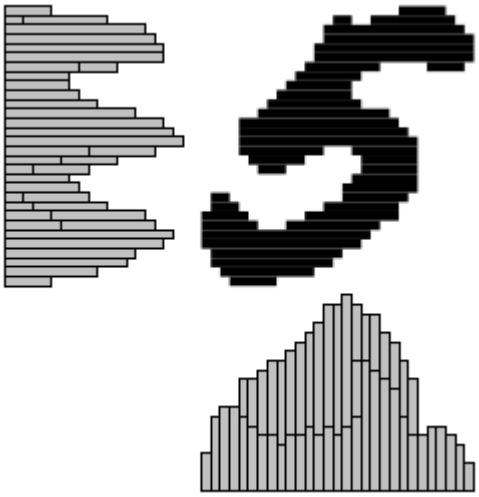


Figure 3: Horizontal and vertical projection histograms [3].

As we have mentioned, there is no method that is intrinsically perfect for a given task. Evaluation of such methods would take a lot of time and it is not in the scope of this work.

Instead, we set our focus on *multilayer feedforward neural networks*, which can be viewed as a combination of a feature extractor and a classifier [3], the latter of which will be explained shortly.

In our work, we have used the *multilayer perceptron* neural network model, which will be more deeply described in the next chapter. For now, we can think of this model as a directed graph consisting of at least 3 layers of nodes.

The first layer is called the *input layer*, the last layer is the *output layer*, and a number of intermediate layers are known as *hidden layers*. Except of the input layer, nodes of neural networks are also called *neurons* or *units*. Each node of a layer typically has a weighted connection to the nodes of the next layer. The hidden layers are important for feature extraction, as they create an internal abstraction of the data fed into the network. The more hidden layers there are in a network, the more abstract the extracted features are.

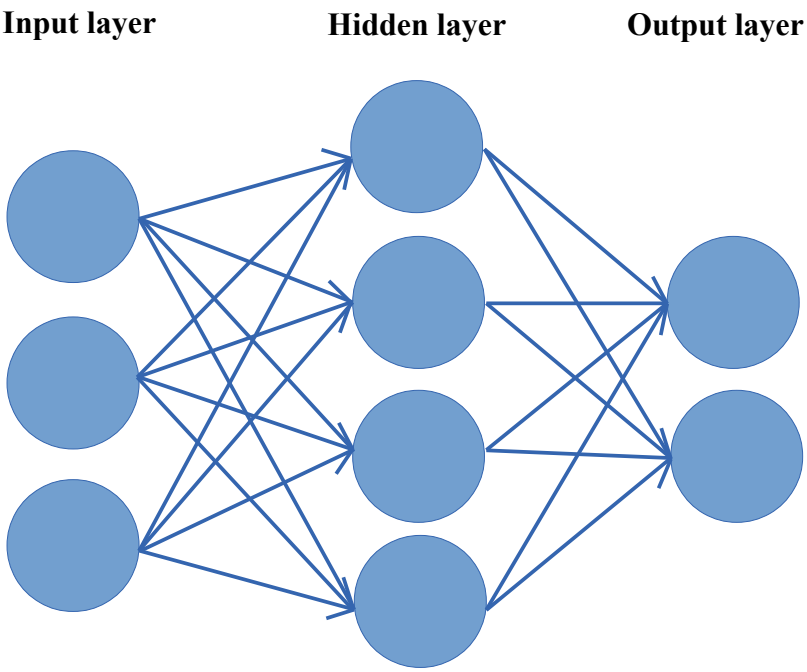


Figure 4: Basic view at the multilayer perceptron architecture. It contains three layers, one of them being a hidden layer. Layers consist of neurons; each layer is fully connected to the next one.

1.2.3 Classification

Classification is defined as the task of assigning *labels* (categories, classes) to yet unseen observations (instances of data). In machine learning, this is done on the basis of training an algorithm on a set of *training examples*. Classification is a *supervised learning* problem, where a "teacher" links a *label* to every instance of data. Label is a discrete number that identifies the class a particular instance belongs to. It is usually represented as a non-negative integer.

There are many machine learning models that implement classification; these are known as *classifiers*. The aim of classifiers is to fit a *decision boundary* (Figure 4) in feature-space that separates the training examples, so that the class of a new observation instance can be correctly labeled. In general, the decision boundary is a hyper-surface that separates an N-dimensional space into two partitions, itself being $N-1$-dimensional.

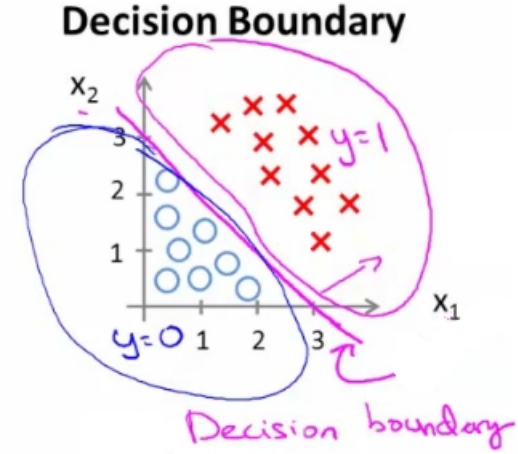


Figure 5: Visualization of a decision boundary. In feature-space given by x1 and x2, a decision boundary is plotted between two linearly separable classes [9].

1.2.3.1 Logistic Regression

Logistic regression is a simple linear classifier. This algorithm tries to find the decision boundary by iterating over the training examples, trying to fit *parameters* that describe the decision boundary hyper-surface equation. During this learning process, the algorithm computes a *cost function* (also called error function), which represents the error measure of its *hypothesis* (the output value, prediction). This value is used for penalization, which updates the parameters to better fit the decision boundary. The goal of this process is to converge to parameter values that *minimize* the cost function. It has been proved [8] that logistic regression is always convex, therefore the minimization process can always converge to a minimum, thus finding the best fit of the decision boundary this algorithm can provide.

Until now, we have been discussing binary classification. To apply logistic regression to handwriting recognition, we would need more than 2 distinguishing classes, hence we need multi-class classification. This can be solved by using the *one-vs-all* approach [9].

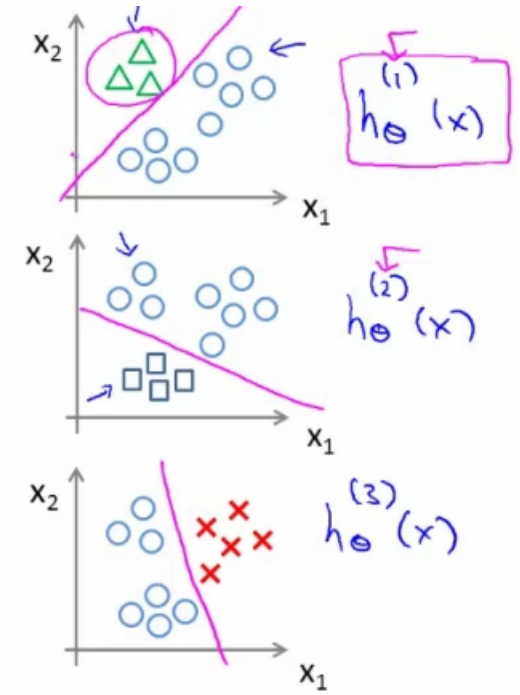


Figure 6: Multi-class classification of three classes as it is split into three sub-problems [9].

The principle of one-vs-all is to split the training set into a number of binary classification problems. Considering we want to classify handwritten digits, the problem degrades into 10 sub-problems, where individual digits are separated from the rest. Figure 6 shows the same for 3 classes of objects. The output of each sub-problem is a probability measure representing how likely a data instance belongs to the class at hand. The overall class is chosen as the class the sub-problem of which has the maximum probability.

To correctly classify data that are not linearly separable, one can use a logistic regression algorithm while using a non-linear hypothesis to separate the classes. One then has to include high-order polynomials in the feature vector to approximate the complex decision boundary. This has an implicit problem of choosing the right polynomials, which can be an enormous task; moreover, this problem is very specific to the task being solved. To avoid this, all of the possible polynomials could be included, but in this way, the number of features grows exponentially and it beings to be computationally challenging to train on the data.

1.2.3.2 Multilayer Perceptron

Multilayer perceptrons (MLPs) are artificial neural networks, learning models inspired by biology. As opposed to logistic regression, which is only a linear classifier on its own, the multilayer perceptron learning model, which we already have mentioned in terms of feature extraction, can also distinguish data that are not linearly separable.

We have already outlined the architecture of an MLP, as seen in (Figure 4).

In order to calculate the class prediction, one must perform *feedforward propagation*. Input data are fed into the input layer and propagated further, passing through weighted connections into hidden layers, using an *activation function.* Hence, the node's *activation* (output value at the node) is a function of the weighted sum of the connected nodes at a previous layer. This process continues until the output layer is reached.

The learning algorithm, *backpropagation*, is different from the one in logistic regression. First, the cost function is measured on the output layer, propagating back to the connections between the input and the first hidden layer afterwards, updating unit weights.

MLPs can perform multi-class classification as well, without any modifications. We simply set the output layer size to the number of classes we want to recognize. After the hypothesis is calculated, we pick the one with the maximum value.

A nonlinear activation function is required for the network to be able to separate non-linearly separable data instances. This, along with the mentioned algorithms, will be explained in next chapters.

1.3 Existing Applications

Handwritten character recognition is currently used extensively in OCR and ICR systems. These are used for various purposes, some of which are listed below.

1.3.1 Full-featured Document OCR and ICR Systems

ABBYY FineReader by the ABBYY company is a piece of software with worldwide recognition that deals with OCR and ICR systems, as well as applied linguistics [11]. The company has also developed a business card reader mobile application that uses a

smartphone's camera for text recognition to import contact information [12]. The application is, among others, also available for the Android platform.

Tesseract-ocr is an OCR engine developed at HP Labs between 1985 and 1995. It is claimed [13] that this engine is the most accurate open source OCR engine available, supporting a wide variety of image formats and over 60 languages. It is free and open source software, licensed under Apache License 2.0.

Google Goggles is an image recognition Android and iOS application, featuring searching based on pictures taken by compatible devices and using character recognition for some use cases [14].

1.3.2 Input methods

Microsoft has been supporting a tablet handwriting-based input method since the release of Windows XP Tablet PC Edition [15]. This allows users of devices with this platform to write text using a digitizing tablet, a touch screen, or a mouse, which is converted into text that can be used in most applications running on the platform.

Google Translate, a machine translation Android application from Google, features handwriting recognition as an input method, as well as translating directly from the camera [16]. This closely resembles a possible extension of our work in the future.

2. Learning Model in Detail

In this chapter, we explain in detail the model that has been used in the project to successfully perform handwritten character recognition. Because this work primarily discusses machine learning methods, the scope is limited to corresponding algorithms. Specifically, as the multilayer perceptron has been chosen as a method for feature extraction and classification, we deal with algorithms required for MLP implementation. In case of insufficiency of the explanation of image preprocessing methods in the overview chapter, refer to [10].

It has been mentioned in the previous chapter that the multilayer perceptron is a supervised learning model that is able to classify multi-class, non-linearly separable data. As such, it is considered one of the most powerful learning algorithms [9].

2.1 Representation

We have previously outlined the architecture of a multilayer perceptron. In short, it is a directed graph with at least three *layers*, at least one of which is a *hidden layer*. The more hidden layers there are, the most abstract features can be extracted from input data and the more complex the decision boundary is. *Neurons* are the vertices of the graph, while *weighted connections* are the edges. Output values of neurons are referred to as *activations*, and to compute the activation of a neuron in a layer, we use an *activation function* on the weighted sum of the connected neuron values in the previous layer.

Input data are usually represented as a feature vector of real values, where the elements are fed to individual input neurons. The representation of output data (hypothesis) depends on the size of the output layer.

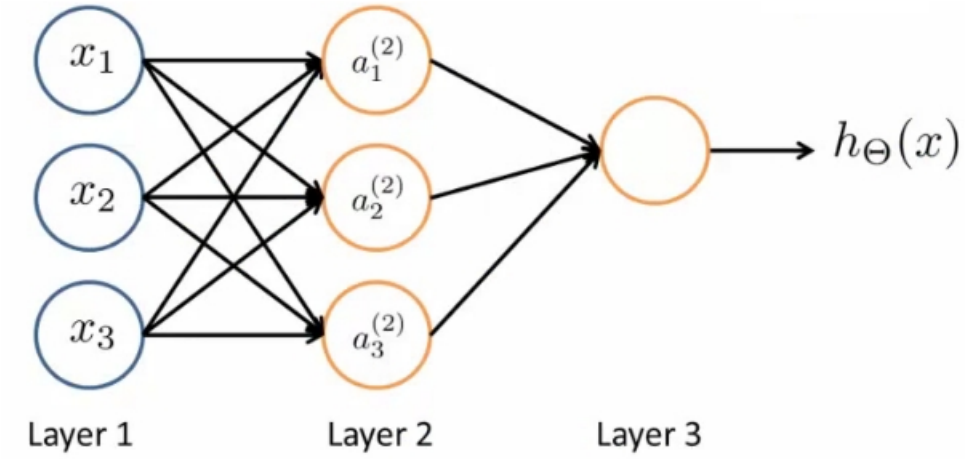


Figure 7: The architecture of a multilayer neural network. Layer 1 is the input layer; layer 2 is a hidden layer; layer 3 is the output layer. x1, x2, and x3 are features fed to the network; a1, a2, and a3 are hidden layer units; $h_\Theta(x)$ is the output value (hypothesis). Each layer is fully connected to the next one [9].

- In binary classification, using a single output neuron is recommended [9]. Here, the hypothesis is typically a real value. A threshold is then used to determine the predicted class.
- In multi-class problems, the size of the output layer is typically equal to the number of classes. Thus, output data is represented as a vector of real values. The predicted class is the element with the maximum value.

In general, however, we assume the output layer is always a real valued vector:

for output layer L,
$$h_\Theta(x) = a^{(L)} = \begin{bmatrix} a_1^{(L)} \\ a_2^{(L)} \\ \dots \\ a_{s_L}^{(L)} \end{bmatrix} \tag{2.1}$$

To perform supervised learning, a set of *labels* (classes) has to be provided. We represent these as a vector of the same size as the output vector:

$$y = \begin{bmatrix} y_1 \\ y_2 \\ \dots \\ y_{s_L} \end{bmatrix} \tag{2.2}$$

The weights of a neuron are represented as real-valued matrices:

for layer $l < L$,
$$\Theta^{(l)} = \begin{bmatrix} \Theta_{1,1}^{(l)} & \Theta_{1,2}^{(l)} & \dots & \Theta_{1,s_l+1}^{(l)} \\ \Theta_{2,1}^{(l)} & \Theta_{2,2}^{(l)} & \dots & \Theta_{2,s_l+1}^{(l)} \\ \dots & \dots & \dots & \dots \\ \Theta_{s_{l+1},1}^{(l)} & \Theta_{s_{l+1},2}^{(l)} & \dots & \Theta_{s_{l+1},s_l+1}^{(l)} \end{bmatrix} \tag{2.3}$$

Here, l is any layer except of the output layer. Using our notation, $\Theta^{(l)}$ is a matrix of weights corresponding to the connection mapping from layer l to layer $l + 1$. s_l is the number of units in layer l, and s_{l+1} is the number of units in layer $l + 1$. Thus, the size of $\Theta^{(l)}$ is $[s_{l+1}, s_l + 1]$.

The additional neuron that is included in layer l is the *bias* neuron. Usually marked as x_0 or $a_0^{(l)}$, bias is a very important element in the network. It can alter the shift of the activation function along the x-axis. The bias neuron is only connected to the next layer—it has no input connections.

Note that a row in the weights matrix represents connections from all of the neurons in layer l to a single neuron in layer $l + 1$. Conversely, a column in the matrix represents connections from a single neuron in layer l to all of the neurons in layer $l + 1$.

2.2 Hypothesis

Throughout this text, we've referred to hypothesis several times. It is the prediction of a class, the output value of a classifier. As mentioned in chapter 1, in order to enable the network to solve complex nonlinear problems, the use of a nonlinear activation function is required.

In many cases, the *sigmoid* activation function is used:

$$g(z) = \frac{1}{(1 + e^{-z})}, \qquad z \in \mathbb{R} \tag{2.4}$$

The range of the sigmoid function is [0, 1], which is therefore also the range of the elements in the output layer.

An activation of a neuron in a layer is computed as a sigmoid function of the linear combination of the weights vector corresponding to the neuron and the activation of all connected neurons from the previous layer. For convenience, we define the input layer neuron vector as

$$x = a^{(1)} \tag{2.5}$$

Using (2.4) and (2.5), we generalize the computation of a neuron's activation in a vectorized form as

for layer $l > 1$, $$a^{(l)} = g\left(\Theta^{(l-1)} a^{(l-1)}\right) \tag{2.6}$$

Here, the sigmoid function is applied element-wise to the product of the weights and the connected neurons from the previous layer, therefore $a^{(l)} \in \mathbb{R}^{s_l}$.

It may be intuitive to go ahead and use (2.6) recursively to compute the overall hypothesis of the network, but as we're assuming the bias neuron in the architecture, it needs to be added to the vector of activations in layer l intermittently.

The process of determining the value of the hypothesis in the described way is called ***forward propagation***. The algorithm broken up into steps follows:

1. Start with the first hidden layer.
2. Compute the activations in the current layer using (2.6).
3. If the current layer is the output layer, we have reached the hypothesis and end.
4. Add a bias unit $a_0^{(l)} = 1$ to the vector of computed activations.
5. Advance to the next layer and go to step 2.

2.3 Learning: Backpropagation

A multilayer perceptron is a supervised learning model. As such, every example in the training set is assigned a label, which is used to compute a cost function (an error measure). As mentioned in the first chapter, learning is an optimization problem that updates free parameters (weights) in order to minimize the cost function.

There is a number of different cost functions typically used when training multilayer perceptrons. Training is often performed by minimizing the mean squared error, which is a sum of squared differences between computed hypotheses and actual labels. However, mean non-squared error is also used. To be consistent with [9], we have used a generalization of the cost function used in logistic regression:

$$J(\Theta)=\frac{1}{m}\sum_{i=1}^{m}\sum_{k=1}^{K}\left[-y_k^{(i)}\log\left(\left(h_\Theta(h^{(i)})\right)_k\right)-\left(1-y_k^{(i)}\right)\log\left(1-\left(h_\Theta(x^{(i)})\right)_k\right)\right] \qquad (2.7)$$

In (2.7), m is the number of training examples and K is the total number of possible labels. $h_\Theta(x^{(i)})$ is computed using forward propagation described above.

The cost function above is not always convex—there may be more local minima. However, according to [9], it is sufficient to reach a local minimum that is not global. In order to reach a local minimum of the cost function, we use an optimization algorithm, such as *gradient descent*. Gradient descent is a simple optimization algorithm that converges to a local minimum by taking steps in a weight-space iteratively in so-called *epochs*. The size of each step is proportional to the negative of the *gradient* of the cost function at the current point.

There is an important factor that modifies the descent step size in machine learning—the learning rate. It is a modifier that is used to tune how fast and how accurate the optimization is and heavily determines the efficiency of a learning algorithm.

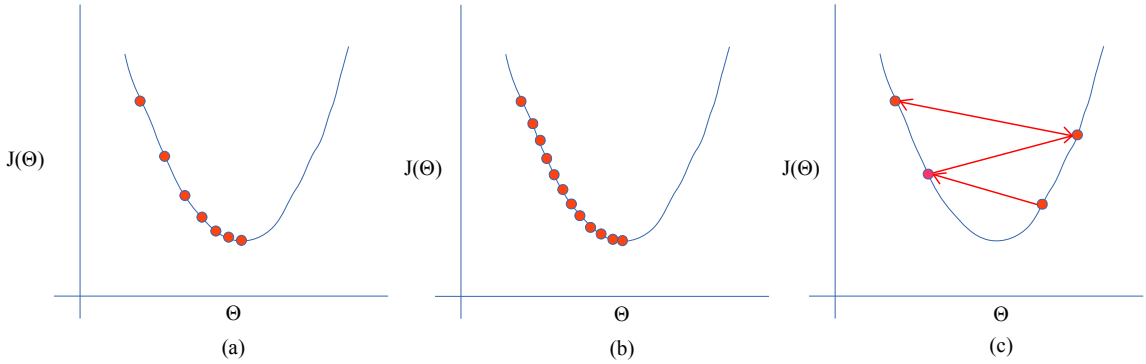


Figure 8: The effect of learning rate on gradient descent. In (a), the learning rate is optimal and gradient descent efficiently converges to a minimum. In (b), the learning rate is too small; the algorithm still converges to a minimum, but takes too many steps. In (c), the learning rate is too large and the algorithm diverges.

Since gradient is the partial derivative of the cost function with respect to individual parameters, the change of parameters in a single gradient descent step is performed as:

$$\Theta = \Theta - \alpha \frac{\partial J(\Theta)}{\partial \Theta} \tag{2.8}$$

In Figure 8 we can see the effect of a wrong choice of the learning rate. There are advanced ways on determining the right learning rate value, but it is usually sufficient to determine it empirically after applying various learning rates to the learning algorithm and picking one with the minimum error.

Obtaining the gradient in a multilayer perceptron is not trivial and is done in several steps.

As each neuron has its own activation and weighted connections, it takes its own part in the cost function. To propagate the error measured on the output layer after a prediction, each neuron's weights need to be updated differently. To achieve this, we introduce the concept of an error term $\delta_j^{(l)}$, representing the error of node *j* in layer *l*.

To obtain the error terms for all neurons in the network except the input layer (as there is no error in the input data), we do the following. Given an instance of input data *x*, forward propagation is performed to determine the hypothesis. With the input label y_j, starting at the end of the network, we calculate the error terms for the output layer per neuron *j*:

$$\delta_j^{(L)} = a_j^{(L)} - y_j \tag{2.9}$$

Note that the output activation $a_j^{(L)}$ is a part of the hypothesis as shown in (2.1).

We then propagate the error to lower layers:

for layer $1 < l < L$, $$\delta_j^{(l)} = (\Theta^{(l)})^T \delta_j^{(l+1)} .* g'(z^{(l)}), \tag{2.10}$$

where g' is the gradient of the sigmoid function and $z^{(l)}$ is a vector of the linear combinations of all neurons and their respective weights in layer l - 1:

for layer $1 < l \leq L$, $$z^{(l)} = \Theta^{(l-1)} a^{(l-1)} \tag{2.11}$$

Using (2.11), it can be shown that the sigmoid gradient is

for layer $1 < l \leq L$, $$g'(z^{(l)}) = a^{(l)} .* (1 - a^{(l)}), \tag{2.12}$$

therefore (2.10) can be expressed using (2.12) as:

for layer $1 < l < L$, $$\delta_j^{(l)} = (\Theta^{(l)})^T \delta_j^{(l+1)} .*(a^{(l)} .*(1 - a^{(l)}))$$ (2.13)

Collecting the error terms is essential for the computation of the partial derivatives of the network. We will now describe the overall process of obtaining the partial derivatives, called ***backpropagation***, in the following pseudo-code. Note that in this pseudo-code, matrix elements are indexed from 0, as opposed to mathematical notation, where indexing starts at 1.

```
01| We are given a training set  {(x^(1), y^(1)), ..., (x^(m), y^(m))}
02| Set  Δ^(l)_{i,j} := 0  for all l, i, j
03| For i = 1 to m
04|     Set  a^(1) := x^(i)
05|     Perform forward propagation to compute a^(l) for l = 2, 3, …, L
06|     Using y^(i), compute  δ^(L) = a^(L) − y(i)
07|     Compute  δ^(L−1), δ^(L−2), ..., δ^(2)
08|     Set  Δ^(l) := Δ^(l) + δ^(l+1) (a^(l))^T
09|  D^(l)_{i,j} := (1/m) Δ^(l)_{i,j}
```

Now, the *D* term is equal to the following:

for layer $l < L$, $$D^{(l)} = \frac{\partial J(\Theta)}{\partial \Theta^{(l)}}$$ (2.14)

Using these partial derivatives, we can perform gradient descent to minimize the cost function and thus enable the algorithm to make predictions on new data.

It is important to note that before using backpropagation to learn the parameters, weights should be initialized to random small numbers between 0 and 1. The weights must not be initialized to zero, otherwise individual weight updates will be constant for all weights and the minimization algorithm will fail to converge to a local minimum.

2.4 Learning: Resilient Backpropagation

Resilient backpropagation (RPROP) is an efficient optimization algorithm proposed by Riedmiller and Braun in 1993 [4]. It is based on the principle of gradient descent used with pure backpropagation. Instead of updating weights of a network with a fixed learning rate

that is constant for all weight connections, it performs a direct adaptation of the weight step using only the sign of the partial derivative, not its magnitude. As such, it overcomes the difficulty of setting the right learning rate value.

For each weight, an individual weight step size is introduced: $\Delta_{i,j}$ (not to be confused with the symbol used when accumulating error in pure backpropagation). Given epoch $t > 0$ and considering Θ as a weight matrix for a single layer (this can be applied to any valid layer), this value evolves during the learning process according to the following rule:

$$\Delta_{i,j}^{(t)} = \begin{cases} \eta^{+} * \Delta_{i,j}^{(t-1)} & , if \frac{\partial J(\Theta)^{(t-1)}}{\partial \Theta} * \frac{\partial J(\Theta)^{(t)}}{\partial \Theta} > 0 \\ \eta^{-} * \Delta_{i,j}^{(t-1)} & , if \frac{\partial J(\Theta)^{(t-1)}}{\partial \Theta} * \frac{\partial J(\Theta)^{(t)}}{\partial \Theta} < 0 \\ \Delta_{i,j}^{(t-1)} & , else \end{cases} \quad (2.15)$$

$$where\ 0 < \eta^{-} < 1 < \eta^{+}$$

The change of the sign of the partial derivative across two epochs indicates the local minimum of the cost function has been jumped over because of a large weight update step. In this case, the step size is decreased by the factor η^{-}. On the other hand, if the sign of the partial derivative has not been changed, the step size is increased by the factor η^{+}. In effect, this method decelerates in gradient descent when the step would be too large to converge, and accelerates in shallow regions where the step size is too small.

After determining the update values as shown in (2.15), the weights are updated, following a rule:

$$\Delta\Theta_{i,j}^{(t)} = \begin{cases} -\Delta_{i,j}^{(t)} & , if \frac{\partial J(\Theta)^{(t)}}{\partial \Theta} > 0 \\ +\Delta_{i,j}^{(t)} & , if \frac{\partial J(\Theta)^{(t)}}{\partial \Theta} < 0 \\ 0 & , else \end{cases} \quad (2.16)$$

$$\Theta^{(t+1)} = \Theta^{(t)} + \Delta\Theta_{i,j}^{(t)} \quad (2.17)$$

In words, if the derivative is positive, which indicates an increasing error, the weight is decreased; if the derivative is negative, the weight is increased.

An initial value of the weight update step Δ_0 has to be set before using the algorithm. This value is preferably chosen in proportion to the initial weight size. However, it has been

shown [4] that the choice of this value is not critical.

Empirically, the reasonable values for η^-, η^+, and Δ_0 are 0.5, 1.2, and 0.1, respectively [4].

In the original document [4] it was suggested that the previous update step be reverted if the sign of the gradient changes (the minimum was missed). This is called *backtracking*. However, in [5], an RPROP variation without backtracking has been proposed, simply leaving this step out, as it is not crucial in the optimization process and is easier to implement.

2.5 Bias, Variance

High ***bias*** and high ***variance***, also called ***underfitting*** and ***overfitting***, respectively, are common causes of an unsatisfying learning algorithm performance.

If a learning model is presented with input with too many complex features, the learned hypothesis may fit the training set very well ($J(\Theta) \approx 0$), but may fail to generalize the prediction on new examples. In this case, overfitting may be observed. On the other hand, if a classifier generalizes the hypothesis to an overly simple form, the error is usually high on both the training set and new examples, which is caused by underfitting.

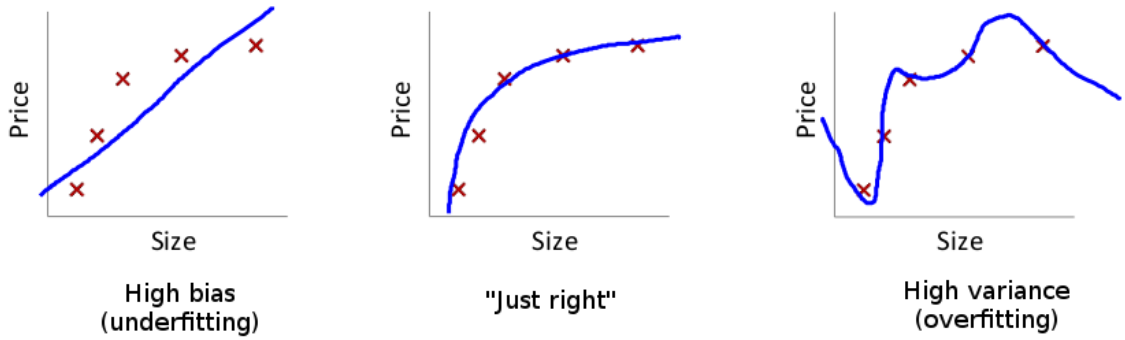


Figure 9: High bias, high variance. On the left, a learning algorithm underfits the training examples and will likely fail at predictions on unseen examples; in the center, the algorithm fits the examples „just right"; on the right, the algorithm overfits the examples, fitting the training set, but will likely fail at predictions on unseen examples.

To address overfitting, we may do one of the following:

1. Reduce the number of features
    - Manually select which features to keep
    - Use a dimensionality reduction algorithm, such as *principal component analysis* (not covered in this work)
2. Get more examples (may help in some cases)
3. Apply *regularization*
    - Decreases the values of free parameters for better generalization
    - Used when all features contribute to a successful hypothesis

In regularization, a regularization parameter λ is used to penalize free parameters (weights in an MLP). This is a real scalar value that takes part in the cost function and optimization functions in order to affect the choice of free parameters and help with the high variance problem. In the learning process, when $\lambda > 0$, the machine learning algorithm's parameters are reduced, when $\lambda < 0$, the parameters are increased, and when $\lambda = 0$, no regularization is performed. Being stated, we choose a high value for the regularization parameter to avoid high variance and lower it in the case of high bias, because setting a too large regularization parameter may be the cause of high bias itself.

To add regularization to the multilayer perceptron algorithms, we must modify (2.7):

$$J(\Theta)=\frac{1}{m}\sum_{i=1}^{m}\sum_{k=1}^{K}\left[-y_k^{(i)}\log\left(\left(h_\Theta(h^{(i)})\right)_k\right)-\left(1-y_k^{(i)}\right)\log\left(1-\left(h_\Theta(x^{(i)})\right)_k\right)\right] + \frac{\lambda}{2m}\left[\sum_{l=1}^{L-1}\sum_{i=1}^{s_{l+1}}\sum_{j=2}^{s_l}\left(\Theta_{i,j}^{(l)}\right)^2\right] \tag{2.18}$$

In effect, this adds a condition to line 9 of the backpropagation pseudo-code:

09| $D_{i,j}^{(l)}:=\frac{1}{m}\Delta_{i,j}^{(l)}$ `if j = 0`

10| $D_{i,j}^{(l)}:=\frac{1}{m}\left(\Delta_{i,j}^{(l)}+\lambda\Theta_{i,j}^{(l)}\right)$ `if j ≠ 0`

As shown in (2.18) and in line 9, we do not add the regularization term for the bias units, therefore we skip the first column of the weight matrix.

3. Solution Description

3.1 Functional Specification

As mentioned in the aim, there are four main requirements of the Android application that has been developed:

- ability to recognize characters written using the touch-sensitive display,
- ability to recognize characters given an image or camera frames as input,
- ability to learn progressively based on user's feedback
- ability to learn new, previously unseen characters.

We will now describe these requirements in detail, listed below.

[R01] The application provides means to enter a “touch mode” screen. Here, the user can draw characters freely by hand.

[R01.1] When the user is done drawing, the drawing is recognized and the prediction is shown to the user.

[R01.2] The drawing, along with the predicted label, can be saved to a persistent location.

[R01.3] The user can provide feedback on the prediction, signaling whether the prediction was correct, or making a correction to the prediction, performing online learning.

[R02] The application provides means to enter a “camera mode” screen. Here, the device's camera is used to present its input to the screen.

[R02.1] After showing the camera frame, it is analyzed and found patterns are recognized and shown to the user.

[R02.2] The process in [R02] and [R02.1] is done continuously as the camera frames are updated over time.

[R02.3] The user can provide feedback on the prediction, signaling whether the prediction was correct, or making a correction to the prediction, performing online learning.

[R03] The application provides means to enter an “image mode” screen. Here, the user

can load an image file present on the device, which is then shown on the screen.

[R03.1] After showing the image, it is analyzed and found patters are recognized and shown to the user.

[R04] The user can see the list of all learned characters.

[R05] The user can add a new character and train it using touch mode as described in [R01].

[R06] The user can perform offline learning on current persistent data at any time.

3.2 Plan of Solution

Here we are going to describe the choices that were needed to be made before starting to implement the solution.

3.2.1 Recognition Process Pipelines

So far, we have named and described several algorithms dealing with image preprocessing, learning, and optimization. These are thought of as building blocks when designing the recognition process pipeline that would satisfy given requirements and they are often used in sequence, one providing input for another. As briefly mentioned in the overview, we have considered a process pipeline for recognition based on touch input, and a separate pipeline for recognition based on static image input.

The recognition pipeline used to recognize characters in the touch input mode follows:

1. Acquire the handwritten character image as a gray-scale bitmap.
2. Resize this bitmap to 20x20 pixels.
3. Acquire a binary bitmap of points where each stroke has started and ended.
4. Resize this bitmap to 20x20 pixels.
5. Unroll the bitmap matrices to a feature vector of 800 elements.
6. Feed this vector to a trained multilayer perceptron, giving us the prediction.

The bitmaps have been chosen to be gray-scale and have a small resolution because it is sufficient to perform correct prediction and the feature vector size is small enough to make

the learning and prediction computationally feasible. Similar bitmap sizes have been chosen in related problems by Ng [9] and LeCun et al. [1].

As we have more data available in the touch mode than a pure image bitmap, we have also decided to collect the bitmap of stroke end points to be able to better distinguish characters such as '8' and 'B', as mentioned in the overview. The resized bitmaps of these characters are often similar, but the writing style of each is usually different. By providing this extra bitmap with each example, we are giving a hint to the neural network classifier about what features to focus on when performing automatic feature extraction with the hidden layer.

The pipeline for recognition based on an image or a camera frame is different:

1. Acquire the image bitmap in gray-scale colors.
2. Apply a median filter to the bitmap.
3. Segment the bitmap using thresholding to get a binary bitmap.
4. Find the bounding boxes of external contours in the bitmap.
5. Extract sub-bitmaps from the bounding boxes.
6. Resize the sub-bitmaps to 20x20 pixels.
7. Unroll the sub-bitmap matrices to feature vectors per 400 elements.
8. Feed each feature vector to a trained multilayer perceptron, giving us predictions.

Our intention is to produce a similar input for the network as in the case of touch input, only without the stroke end bitmaps as we do not possess this information in this mode. With this approach, we are able to re-use bitmaps produced in touch input to train a network that operates on images.

When preprocessing the image, we apply a median blur to remove noise. Median blur was preferred to other blur filters because of its property of preserving edges, which are important in character recognition.

Thresholding is a very simple segmentation algorithm that is sufficient to segment out characters from the background and thus binarize the bitmap. This step is required, as the background is an extra information we do not need in the recognition process.

Finding contours in the image and finding the bounding boxes of each pattern allows us to

detect more characters in an image at once. Given the bounding boxes, we are able to extract bitmaps of the individual characters (or other patterns) and then perform the rest of the processes just as in the touch mode pipeline, excluding, of course, the stroke end maps in the feature vectors.

3.2.2 Network Architecture

We have used a multilayer perceptron as a feature extractor and a classifier. The model of three layers, as used by Ng in [9], with the hidden layer size of 25, has been a likely candidate. The choice of the right architecture determines the tendency of overfitting and underfitting, while also affecting the computational performance of learning and predictions. The results of implementation will be discussed in chapter 5.

As input, we will be considering only a set of digits (the network will distinguish 10 classes). However, the application will be ready to learn new characters, such as alphabetical, or even punctuation marks.

It has already been mentioned that the feature vector size is 800 for touch mode and 400 for image and camera mode. Therefore, two different initial architecture instances are maintained. In touch mode, the input, hidden, and output layer sizes are 800, 25, and 10, and in image/camera mode, they are 400, 25, and 10, respectively. The output layer sizes are dynamic, as they change when new characters are learned.

Typically, a large number of training examples is used. This is usually proportional to the input layer size. In our example, it may generally seem reasonable to train the networks on thousands of examples. However, we have designed the system to learn progressively and save new examples after prediction. Also, every user has a different handwriting style, so collecting a large number of examples would require to collect data from many sources. In order to simplify this, we have only provided 40 examples per digit in the initial training set, leaving it up to the user to produce new examples. The application facilitates this process.

3.2.3 Offline and Online Learning

Offline and online learning have already been defined in the overview chapter as eager and lazy leaning mechanisms, respectively. As the specification denotes, both mechanisms have to be implemented in our application in order to be able to learn progressively based on user's feedback, as well as perform offline learning on persistent data at any time.

The preferred algorithm for offline learning is RPROP because of the described advantages over pure backpropagation. It is evident the algorithm is superior to backpropagation and other learning algorithms as shown in (Figure 10).

Average number of required epochs				
Problem	10-5-10	12-2-12	9 Men's Morris	Figure Rec.
BP (best)	121	> 15000	98	151
SSAB (best)	55	534	34	41
QP (best)	21	405	34	28
RPROP (std)	30	367	30	29
RPROP (best)	19	322	23	28

Figure 10: Average number of required epochs in different learning problem scenarios. It is apparent that RPROP is superior to backpropagation (BP), as far fewer epochs are required to converge to a local minimum of the cost function [4].

However, RPROP is not an online learning algorithm—it wouldn't make sense to perform as single learning iteration, because no weight update acceleration or deceleration occurs. In effect, using RPROP for online learning would be equal to using backpropagation if initial weight update value of RPROP is equal to the learning rate of backpropagation. Therefore, pure backpropagation has been chosen to perform online learning.

For backpropagation, we have set the learning rate to 0.3. For RPROP, the initial weight update size is 0.01, η^- has been set to the traditional value of 0.5, and η^+ is equal to 1.2. After testing the performance of the algorithms, we have decided not to use regularization at all (the regularization parameter is 0). Both backpropagation and RPROP perform 100 optimization epochs.

3.2.4 Used Technologies

We have developed a native Android application using Java SE Runtime Environment 6, the Java programming language, and Android SDK. This is the traditional way to build Android applications and does not rely on a third party. The application has been targeted at Android 4.2.2, however, Android 2.2 and up should be supported.

Having explained the algorithms used in multilayer perceptrons, we have decided to implement machine learning on matrix-level only using a library for matrix manipulations, avoiding existing machine learning libraries. For this, we have used a subset of the JAMA Java package that deals with matrices.

Because image preprocessing is not the primary focus of this work, we haven't implemented these algorithms from scratch—instead, we have used the OpenCV library for Android. This library contains algorithms for computer vision in general. Omitting its machine learning capabilities, we have used this to perform image preprocessing steps in the image and camera modes of the application.

To prototype, test, and evaluate algorithms, we have used the GNU Octave language. This approach allowed us to produce statistics and plots useful for configuring the network architecture and debugging the algorithms.

4. Implementation

In this chapter, we will describe how the Android application requirements have been satisfied, document the implementation, describe the problems we've run into and how they were solved and provide a brief user guide in the process.

4.1 Android Application Implementation

The main entry point of the application is a navigation *activity*. Being one of four main building blocks of Android applications, activities are single, focused things that the user can do [17]. They usually appear as windows containing user interface elements that the user can interact with. Our navigation activity contains buttons to access other activities—input mode, camera mode, image mode, and character list. Thus, this activity is required in order to be able to satisfy R01, R02, R03, and R04.

The touch mode activity is designed to implement features required in R01. The main piece of user interface it contains is a custom *view* object that facilitates the drawing of characters. A view is any single piece of user interface that can be shown to the user. This view records a path made of individual strokes the user makes and a stroke end point bitmap as described in the solution plan. It also contains two buttons: “Clear” and “Predict”. The clear button simply clears the drawing, while the predict button initiates the input mode recognition pipeline. The character's bounding box is marked with a rectangle and the prediction is shown to the user in form of a dialog. This dialog presents three buttons: “Cancel”, “Correction”, and “OK”. Cancel intuitively closes the dialog and doesn't interact with the neural network at all. When the correction button is clicked, the dialog asks the user for the correct label in case the drawing has been misclassified. This initiates the online learning algorithm, where backpropagation is performed on this single example, only performing one iteration. The label provided by the user is used in the learning process. The “OK” button does the same, but instead of asking the user for the correct label, the predicted label is used.

Each time the activity goes into background, or the system decides to destroy it, the perceptron state is saved to external storage. This covers requirements R01.1 and R01.3.

By default, each time a prediction is made, the drawing and the stroke end bitmaps are saved to external storage as image files in pairs, so that perceptron weights can be removed and offline learning can be performed. This behavior satisfies R01.2 and can be turned off in application settings to save memory when no or only online learning is required.

The camera mode contains an OpenCV user interface element tailored for showing and processing camera frames in Android. When the user enters the activity, they are immediately presented with continuously updated camera images, which are processed according to the image mode recognition pipeline as has been described. Individual contours that are likely to be characters are found and marked with rectangles. A predicted label is shown above each such rectangle. In practice, a Nexus S device with a single-core 1GHz CPU (the GPU is, unfortunately, not used by the implementation of the OpenCV library which would have likely performed better) and a camera resolution of 720x480 pixels, the updating frequency is approximately 2 frames per second. This depends on the number of objects visible in the frames. In order to filter out contours with a low probability of correct classification, contours that cover an area of less than 40 pixels are not further processed. This significantly improves the framerate, as there are usually many objects in the image that are too large to be filtered by the median blur, but still unwanted. In this mode, the user can click on the individual characters outlined by the mentioned rectangles in order to provide feedback. The feedback dialog described in the touch mode is reused. Corrections of the predictions are immediately visible in the next frames; as such, this activity may resemble an application of augmented reality. As with the touch mode, the neural network state is also saved when the activity goes into background or is killed by the system.

When implementing the camera mode activity, a problem with the choice of segmentation threshold value arose. OpenCV provides the use of *Otsu's method*, which automatically determines the threshold value. However, this hasn't proved to be useful, as it is computationally more expensive and the choice of threshold values still hasn't been optimal. We have realized the best way to address this problem would be to leave the setting of the threshold value to the user, being equal to 100 (in the range 0-255) by default. For this, we have used a panel with a segmentation check-box that allows users to see the segmented image, and a *seekbar* (a UI element for contiguous value selection) to control the threshold.

When the user enters the image mode, they are instructed to start by opening an image file. The system then shows all possible applications that can handle this request; a default gallery application is preferred, as these usually implement the behavior of providing data correctly, however, other applications, such as file managers, may also be used. The loaded image file is processed according to the image mode recognition pipeline using the same techniques as in camera mode, and is then shown to the user. The segmentation control views are also present. Because this activity would share a lot of source code with the camera activity, methods that could be reasonably separated from the classes have been moved to a public class and declared as static.

Unlike in the camera activity, which is constrained to the landscape screen orientation, we wanted to make this activity flexible. It can be rotated, while its state is retained in a *fragment*, which represents a portion of a user interface or a behavior of an activity [17].

The last activity from the four navigation elements is a character list that satisfies R04. It shows a *grid view* of learned characters, filling the items with saved bitmaps if present. When the user selects an item in the grid view, they are taken into the touch mode activity that is modified to focus on training the single character label. The difference is that no feedback dialog is shown to the user—it is assumed that the character the user selected in the character list is always the anticipated label and online learning is performed after each prediction. Otherwise, the activity is unchanged; the examples are still saved to external storage if set to do so in the settings and the network state is saved in appropriate events.

The character list activity also contains a *menu item* for adding a new character as required in R05. After being selected, the user is asked to assign a label to the new character and is then taken into the touch input activity to train the new character as described above. This action adds the character to the list of known characters and modifies the structure of the perceptron used for touch input recognition. In particular, the output layer size is increased by one (to be able to predict the new class) and weight connections to this layer are adjusted and initialized to random values.

Because the application maintains two separate perceptrons, the one used for image and camera recognition is also modified in this way, but only when the user navigates to the related activities. The online learning that was performed when training the new character using touch input is, however, not applied to this perceptron. The user can only train it

using online learning in the related activities, or by performing offline learning in the case that the bitmaps for the new character have been saved.

So far, we have mentioned the application settings only in relation to the option of saving character bitmaps. Settings are another activity that the user can access from any other activity at any time. Here, besides the saving option, the user is able to pick the error minimization algorithm used for offline learning: backpropagation or RPROP, the latter being the default. The initial learning rate to perform the online learning iteration (using backpropagation) is 0.3, but if the user feels the need to adjust this parameter, they may do it here. Finally, the settings contain items that initiate the offline learning process for touch-based and image-based input recognition that satisfy R06.

To make it easy for a user to start using the application, and out-of-the-box experience is required. The application package contains initial files with a set of image bitmaps, saved perceptron weights and the list of known characters. These files are compressed in a single zip file that the application extracts and copies to external storage. To counter the possibility of injecting malicious files in the zip file, its *SHA-1 checksum* is verified before the file is processed.

As the android development guidelines dictate [17], long operations such as file input and output or offline machine learning should be executed in separate threads. It is preferred to use the *AsyncTask* class, which is a thread wrapper that allows simple communication back to the main application thread. However, when the activity in which an AsyncTask is started is released from memory, nothing stops the system from killing the leftover thread that may be performing a critical operation. To ensure such threads keep running, we have created a *service* for copying the initial files, and another one that handles offline learning of both neural networks. These services have been declared *foreground*, which guarantees the system will only reclaim the memory if it is required to keep the user interface responsive [17].

Additionally, when an Android device runs on its battery, the execution of threads may be interrupted when the device is in sleep mode. As the learning process takes a few minutes to complete, it should continue to run even after the power button of the device is pressed. For this, we acquire a *partial wake lock*. This is a mechanism that controls the power state of the device. When a partial wake lock is acquired, the device's screen and keyboard

backlight are allowed to be turned off, but the CPU is kept on until all partial wake locks have been released [17].

4.2 Android Application Source Code

The application source code is composed of Java classes arranged in packages. Here, we will take a brief look at the packages and describe the classes. The complete source code documentation, along with the source code of the Android application and GNU Octave scripts, see the content included on the enclosed DVD.

The list of packages is as follows:

1. eu.uhliarik.charrec.gui.activities
2. eu.uhliarik.charrec.gui.adapters
3. eu.uhliarik.charrec.gui.dialogs
4. eu.uhliarik.charrec.gui.fragments
5. eu.uhliarik.charrec.gui.models
6. eu.uhliarik.charrec.gui.views
7. eu.uhliarik.charrec.learning
8. eu.uhliarik.charrec.services
9. eu.uhliarik.charrec.utils
10. eu.uhliarik.mlp
11. jama

Package 1, *activities*, contains the activities that have already been described in 4.1.

The *adapters* package contains an adapter class *GridItemAdapter*, which is used to fill the character list grid view.

The *dialogs* package consists of four dialog classes: *AboutDialog*, *ChooseCharacterDialog*, *ExternalStorageDialog*, and *PredictionDialog*. These build the dialog components used throughout the application and maintain listeners for button click events.

The fourth package, *fragments*, contains a *GridViewFragment* class that is required in the implementation of the character list grid view so that its state is retained across activity lifetime. Also, the *ImageCharacterFragment* can be found here, which is used for retaining the state of the image view in the image recognition activity.

The *models* package comprises *CharacterGridItemModel*, which is a class that holds a bitmap and a label of a character used in the character list.

We have created two custom views in the *views* package, namely *CharacterGridView* and *CharacterView*. The *CharacterGridView* class extends *GridView* and adds a few methods to facilitate the use of the class. *GridView* extends the *View* class and is the main component of the touch input mode. Its main task is to override the *onDraw* and *onTouchEvent* methods to interact with the user and allow them to draw on the view's canvas.

The *eu.uhliarik.charrec.learning* package contains the *CharacterMlp* class that extends a *MultilayerPerceptron* in the *eu.uhliarik.mlp* package and is used throughout the application as the main learning model.

The Android services described earlier in this chapter can be found in the *eu.uhliarik.charrec.services* package: *TrainingService* and *TransferService*. The former performs offline learning of the neural networks, while the latter manages transfer of the initial data when the application is used for the first time.

Code that would otherwise have been shared by several classes has been separated to the *FileUtils* and *ImageUtils* classes in the *eu.uhliarik.charrec.utils* package. Intuitively, *FileUtils* contains public static methods that deal with saving and loading perceptron data, extracting files from the compressed initial data file, and more. *ImageUtils*, on the other hand, deals with loading and converting the dataset (character bitmaps), bitmap manipulations, bitmap matrix format conversions, and image preprocessing algorithms.

Package 10 is placed outside of the *eu.uhliarik.charrec* package, as it is meant to be used by any application, not only in our project. It consists of classes that implement the multilayer perceptron in a loose coupling fashion. The main class, *MultilayerPerceptron*, can be used alone, but it is suggested that it be extended by a class that is designed to solve a specific problem. The *CharacterMlp* class mentioned in package 7 is an example of such implementation.

Finally, *jama* is an external package containing the *Matrix* class that provides a referential implementation of matrix manipulations. We have modified this class, adding new methods required when implementing the learning algorithms described in chapter 2.

5. Results

Before implementing the learning algorithms in Java, we have tested them as prototypes using GNU Octave. This allowed us to easily collect results of how well the learning algorithms perform. This chapter presents and discusses the comparison of the collected results.

5.1 Collection Methods

We have measured the performance of the algorithms used in multilayer perceptrons: backpropagation and resilient propagation. We have considered the scenario of recognition from image, where the dataset consists only of 40 character image bitmaps per character. For this comparison, the datasets are only comprised of characters of digits, therefore the size of the dataset contains 400 examples.

For relevant values, we have split the dataset into training and validation sets, with the ratio being 7:3. Also, before using the learning algorithms, the dataset has been randomly shuffled.

The error rates have been obtained using a logarithmic cost function shown in (2.7).

The configuration of the learning model whose results are presented here is:

- The regularization parameter is 0.
- The number of epochs is 100.
- In backpropagation, the learning rate is 0.3.
- In resilient backpropagation, η^- , η^+ , and Δ_0 are 0.5, 1.2, and 0.01, respectively.
- The perceptron architectures are as described in the plan of solution.

This configuration has been chosen based on recommendations from various resources that have been referred to in the explanation of individual model characteristics, and testing, the results of which would be too long to fit in this chapter.

5.1 Result Comparison

We have measured the error of the backpropagation and RPROP algorithms on the training and validation sets. This has been tested using fractions of the dataset of various sizes and a learning curve has been plotted. Learning curve represents error as a function of the dataset size and is a perfect tool to visualize high bias or variance.

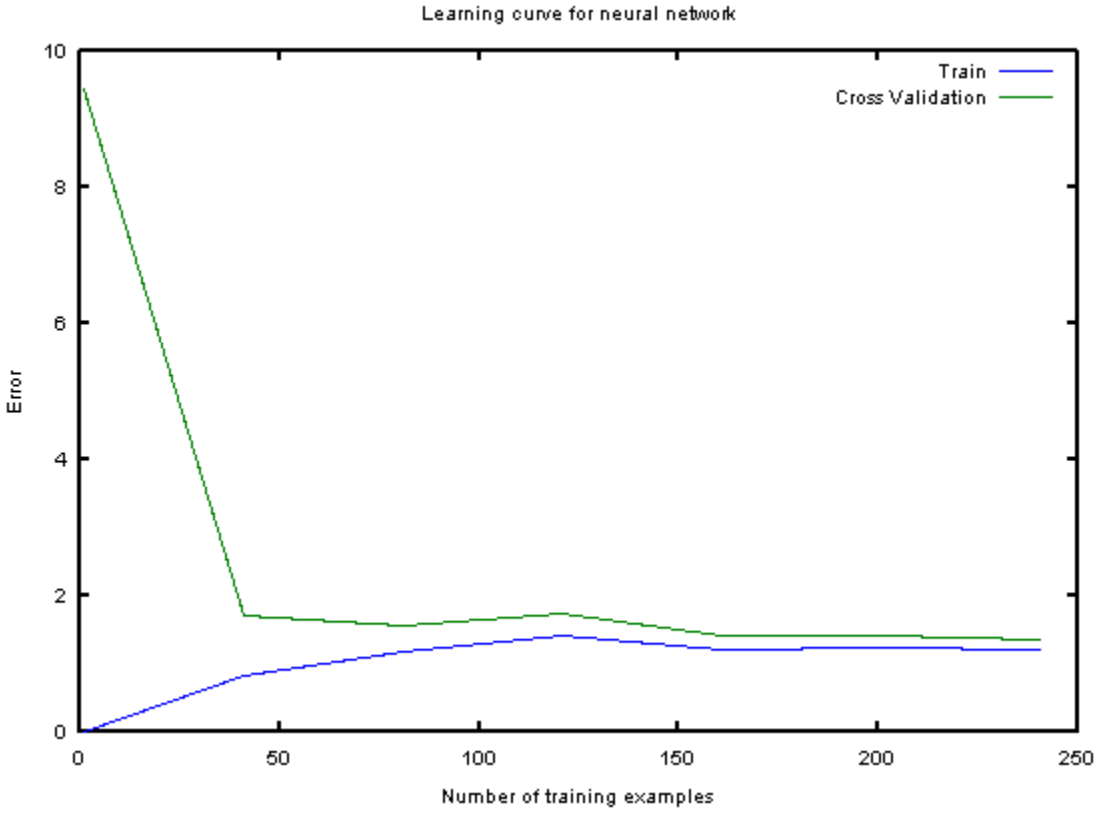


Figure 11: Learning curve of backpropagation performed on the training and validation sets.

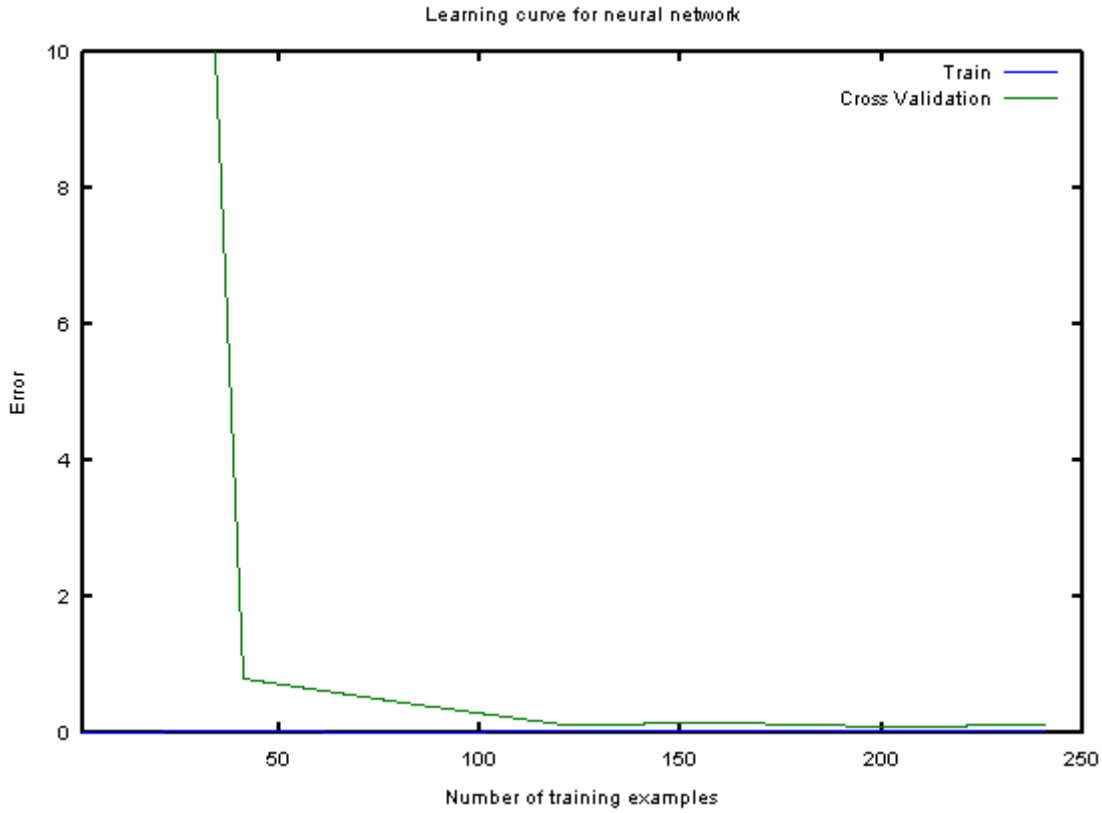


Figure 12: Learning curve of RPROP performed on the training and validation sets.

In the learning curves, no significant overfitting or underfitting is apparent. We can see that the RPROP algorithm manages to converge to a better minimum given 100 epochs than backpropagation. This is caused by the advantages of the RPROP algorithm to pure backpropagation that we explained earlier in this work. Table 1 confirms these findings.

Algorithm	Training set error (m=400)	Validation set error (m=400)	Avg. training set error	Avg. validation set error	Training set accuracy
Backpropagation	1.201	1.396	1.177	1.529	96.25%
RPROP	0.020	0.320	0.019	0.331	100.00%

Table 1: Comparison of learning algorithms; 'm' represents the number of examples used for training. The average error values are measured across the 'x' axis of the learning curve, omitting values where $m \leq 40$, as the validation set error is expected to be high in both algorithms.

6. Conclusion

This work has mostly been focused on the machine learning methods used in the project. At first, we reviewed the approaches that are nowadays used in similar applications. After that, we delved into the inner workings of a multilayer perceptron, focusing on backpropagation and resilient back, which has been implemented an the Android application. With the knowledge we had described, we specified the requirements of the project and planned the solution. During the development of the application, we ran into a few problems, which, along with the application structure and details, have been described in the implementation chapter. Finally, the results of the implementation of the learning algorithms have been compared.

The Android application performs character recognition based on touch, image, and camera input. We have developed a Java package containing classes that implement the multilayer perceptron learning model, which can also be used in other applications due to its modular design that supports the loose coupling principle. The application itself uses this package in such way.

Several improvements for the application or the learning model used within can be suggested. For example, the feature extraction performed by the neural network could be constrained to operate on more strictly preprocessed data. Also, several classifiers learning on different features could be combined to make the system more robust. Additionally, an unsupervised clustering learning model, such as an ART network [7], could be used on the raw input, whose output would be connected to a multilayer perceptron.

Bibliography

[1] Y. LeCun et al., *Backpropagation Applied to Handwritten Zip Code Recognition*, Massachusetts Institute of Technology, 1989

[2] R. Plamondon et al., *On-Line and Off-Line Handwriting Recognition: A Comprehensive Survey*, IEEE Transactions on Pattern Analysis and Machine Intelligence, vol. 22, no. 1, 2000

[3] Ø. Due Trier, A. K. Jain, T. Taxt, *Pattern Recognition*, vol. 29, no. 4, pp. 641–662, 1996

[4] M. Riedmiller, H. Braun, *A Direct Adaptive Method for Faster Backpropagation Learning: The RPROP Algorithm*, University of Karlsruhe, 1993

[5] M. Riedmiller, *Advanced Supervised Learning in Multi-layer Perceptrons - From Backpropagation to Adaptive Learning Algorithms*, University of Karlsruhe, 1994

[6] M. H. Glauberman, *Character recognition for business machines*, Electronics, vol. 29, pp. 132–136, 1956

[7] Ľ. Malinovský. 2007. *Model ART neurónovej siete na kategorizáciu vstupov* : bachelor's thesis. Bratislava : Comenius University, 2007

[8] J. D. M. Rennie, *Regularized Logistic Regression is Strictly Convex*, Massachusetts Institute of Technology, 2005

[9] A. Ng, *Machine Learning—Course Notes*, http://opencourseonline.com/51/stanford-university-machine-learnig-class-lecture-slides-by-professor-andrew-ng, accessed 20. 5. 2013

[10] M. Ftáčnik, *Základy spracovania obrazu—Course Notes*, http://www.sccg.sk/~ftacnik/imageproc.htm, accessed 1. 4. 2013

[11] *ABBYY Company Overview,* http://web.archive.org/web/20091215111857/http://www.abbyy.com/company, accessed 21. 5. 2013

[12] *ABBYY Business Card Reader*, http://www.abbyy.com/products/bcr/, accessed 21. 5. 2013

[13] *Tesseract OCR*, http://code.google.com/p/tesseract-ocr/, accessed 21. 5. 2013

[14] *Google Goggles Mobile Application*, http://www.google.com/mobile/goggles/#text, accessed 21. 5. 2013

[15] *Tablet PC Recognizer Pack, frequently asked questions*, http://support.microsoft.com/kb/828729, accessed 21. 5. 2013

[16] Google Translate Android Application, https://play.google.com/store/apps/details?id=com.google.android.apps.translate, accessed 21. 5. 2013

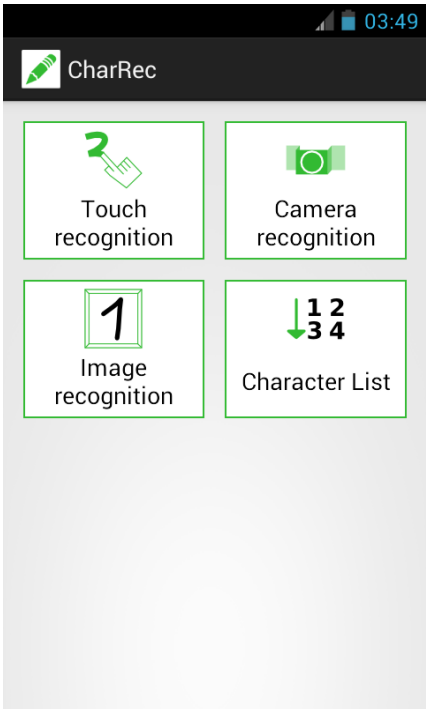
[17] Android Develoepers API Guides, http://developer.android.com/guide/components/index.html, accessed 30. 5. 2013

Appendices

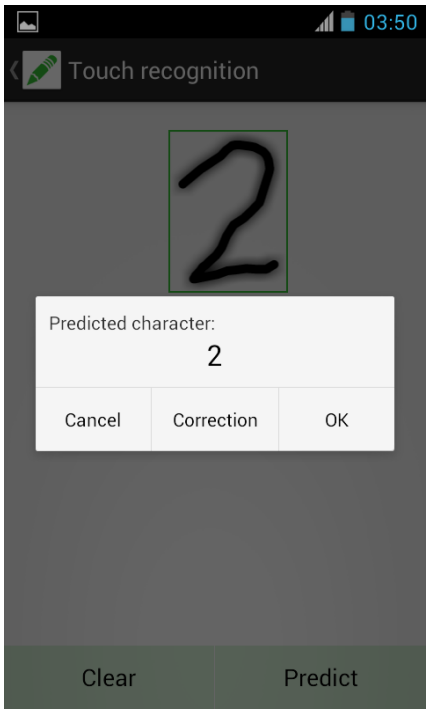
Appendix A: DVD

The enclosed DVD contains an electronic copy of this text, source codes of the Android application and GNU Octave scripts, Java source code documentation of the Android application, and an Android package for installation of the Android application.

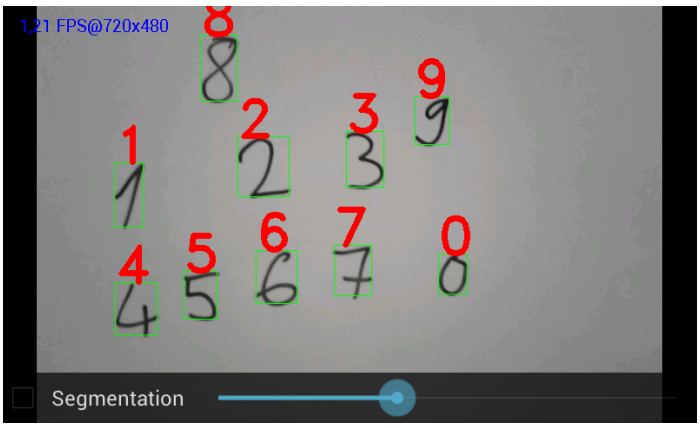
Appendix B: Android Application Screenshots



Screenshot 1: Android application navigation screen



Screenshot 2: Android application touch mode screen



Screenshot 3: Android application camera mode screen