Department of Computing and Information Systems,
The University of Melbourne

Regularization Methods for Neural Networks and Related Models

Sergey Demyanov

Submitted in total fulfilment of the requirements
of the degree of Doctor of Philosophy

Produced on archival quality paper

September, 2015

Abstract

Neural networks have become very popular in the last few years. They have demonstrated the best results in areas of image classification, image segmentation, speech recognition, and text processing. The major breakthrough happened in early 2010s, when it became feasible to train deep neural networks (DNN) on a GPU, which made the training process several hundred times faster. At the same time, large labeled datasets with millions of objects, such as ImageNet [16], became available. The GPU implementation of a convolutional DNN with over 10 layers and millions of parameters could handle the ImageNet dataset in just a few days. As a result, such networks could decrease classification error in the image classification competition LSVRC-2010 [54] by 40% compared with the hand-made feature algorithms.

Deep neural networks are able to demonstrate excellent results on tasks with a complex classification function and sufficient amount of training data. However, since DNN models have a huge number of parameters, they can also be easily overfitted, when the amount of training data is not large enough. Thus, regularization techniques for neural networks are crucially important to make them applicable to a wide range of problems. In this thesis we provide a comprehensive overview of existing regularization techniques for neural networks and provide their theoretical explanation.

Training of neural networks is performed using the Backpropagation algorithm (BP). Standard BP has two passes: forward and backward. It computes the predictions for the current input and the loss function in the forward pass, and the derivatives of the loss function with respect to the input and weights on the backward pass. The nature of the data usually assumes that two very close data points have the same label. This means that the predictions of a classifier

should not change quickly near the points of a dataset. We propose a natural extension of the backpropagation algorithm that minimizes the length of the vector of derivatives of the loss function with respect to the input values, and demonstrate that this algorithm improves the accuracy of the trained classifier.

The proposed invariant backpropagation algorithm requires an additional hyperparameter that defines the strength of regularization, and therefore controls the flexibility of a classifier. In order to achieve the best results, the initial value of this parameter needs to be carefully chosen. Usually this hyperparameter is chosen using a validation set or cross-validation. However, these methods might not be accurate and can be slow. We propose a method of choosing the parameter that affects a classifier flexibility and demonstrate its performance on Support Vector Machines. This method is based on the Akaike Information Criterion (AIC) and Bayesian Information Criterion (BIC), and uses the disposition of misclassified objects and VC-dimension of the classifier.

In some tasks, data consists of feature values as well as additional information about feature location in one or more dimensions. Usually these dimensions are space and time. For example, image pixels are described by their coordinates in horizontal and vertical axes. Various time series necessarily contain information when their elements were taken. This information might be used by a classifier. Some regularizers are particularly targeted at this goal, restricting a classifier from learning an inappropriate model. We present an overview of such regularization methods, describe some of their applications and provide the result of their usage.

Video is one of the domains where one has to consider time. One of the challenging tasks of this domain is deception detection from visual cues. The psychology literature indicates that people are unable to detect deception with high accuracy and is slightly better than a random guess. At the same time, trained individuals were shown to be able to detect liars with an accuracy up to 73% [24, 25]. This result confirms that visual and audio channels contain enough information to detect deception. In this thesis we describe an automated multilevel system of video processing and feature engineering based on facial movements. We demonstrate that the extracted features provide a classification accuracy that is statistically significantly better than a random guess.

Another contribution of this thesis is the collection of one of the largest datasets of videos with truthful and deceptive people recorded in more natural conditions than others.

Declaration

This is to certify that

1. the thesis comprises only my original work towards the degree of Doctor of Philosophy except where indicated in the Preface,
2. due acknowledgment has been made in the text to all other material used,
3. the thesis is fewer than 80,000 words in length, exclusive of tables, maps, bibliographies and appendices.

Sergey Demyanov

Preface

This thesis has been written at the Department of Computing and Information Systems, The University of Melbourne. The major parts of the thesis are the Chapters 4, 5 and 6. They are based on the following publications:

1. Invariant backpropagation: how to train a transformation-invariant neural network. Sergey Demyanov, James Bailey, Ramamohanarao Kotagiri, and Christopher Leckie. arXiv preprint arXiv:1502.04434 (2015). To be submitted to the International Conference of Learning Representations 2016 (Chapter 4 contains material for this publication).

2. AIC and BIC based approaches for SVM parameter value estimation with RBF kernels. Sergey Demyanov, James Bailey, Kotagiri Ramamohanarao and Christopher Leckie. Proceedings of the Fourth Asian Conference on Machine Learning (ACML), pages 97-112, Singapore, 4-6 November, 2012 (Chapter 5 contains material for this publication).

3. Detection of deception in the Mafia party game. Sergey Demyanov, James Bailey, Ramamohanarao Kotagiri and Christopher Leckie. To appear in Proceedings of the 17th ACM International Conference on Multimodal Interaction, Seattle, Washington, USA, November 9-13, 2015 (Chapter 6 contains material for this publication).

Acknowledgments

PhD study is a long and tough journey. It differs from undergraduate study by its uncertainty, and requires a lot of patience and persistence. Instead, it teaches you to study problems from all angles, and efficiently find the solutions of the problems than nobody have solved before. I am happy that I was able to complete this journey. However, it would not be possible without all those people that I mention below.

First of all, I would like to express my grateful acknowledgment to my supervisors: Professor James Bailey, Professor Rao Kotagiri, and Professor Christopher Leckie for accepting me as a PhD student, believing in my ideas, and supporting me during all my PhD study. I very much appreciate freedom in the choice of research directions that they provided to me. It allowed me to be motivated by my research during all four years. I am especially thankful for the support at the moments when they believed in me more than I did. In particular, I would like to thank James for the financial support and useful connections with IBM research, Rao for the most detailed and helpful discussions, and Chris for the useful links, that allowed me to finish my first article.

I would like to to thank a member of my Advisory Committee Dr. Jeffrey Chan, and its head Professor Rajkumar Buyya for supporting the plan of my research at all milestones.

I would like to thank the University of Melbourne, the CIS Department, and personally the Head of the Department Justin Zobel for providing excellent conditions for cutting edge research.

I would like to thank IBM Research (Australia), and personally Dr. Rahil Garnavi for accepting my candidature for internship, that allowed me to finish writing my thesis in Australia, and join IBM Research after submission.

I would especially like to thank all my colleagues from the Machine Learning group, which became my good friends: Simone Romano, Goce Ristanoski, Jiazhen He, Florin Schimbinschi, Andrey Kan. You were one of the reasons that called me to come to the office every day. I can fairly say that it would impossible to be enthusiastic for all time without you. We have had a lot of great moments at the university and outside it, and I value it.

I would also like to thank all my other colleagues from the Machine Learning group, the Cloud Computing group, and other groups of the CIS departments: Yun Zhou, Yang Lei, Yamuna Kankanige, Miao Kang, Shuo Zhou, Daniel, Mohadeseh Ganji, Vinh Nguen, Anton Beloglazov, Adel Toosi, Amir Vahid, Nikolay Grozev, Deepak Poola, Yali Zhao, Chenhao Qu, Bowen Zhou, Farzad Khodadadi, Pallab Roy, Xi Liang, and many others. I am proud to work with you.

Last but not least, I would like to sincerely thank my mom Larisa Demyanova and my dad Vladimir Demyanov, for supporting me, trusting me, and taking care of me for my entire life.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Neural networks have become a leading tool in machine learning for tasks with a large number of features and complex classification functions. The range of such tasks is quite broad. They include image classification, image segmentation, object detection, image description, speech recognition, text generation and others ([59]). The input for these tasks can be a set of image pixels, tone intensities, or letters. In order to approximate complex functions, neural networks contain a huge number of parameters. For example, a famous AlexNet network for image classification [54] contains over 60 million of weights. Such a huge number requires a lot of training data to prevent overfitting - a situation where a classifier becomes too specific to the samples presented in the training set. AlexNet worked well only because it was trained on the ImageNet database [16] with 1.3 million original images, and over a billion of generated images.

However, such large datasets are not always available. In this situation people either have to use neural networks of a smaller size, which decreases their power, or employ various regularization techniques, that incorporate domain knowledge into the learning process, reducing the possible space of appropriate solutions. For example, convolutional layers (Section 3.4) can be considered as one of them. They provide the knowledge that the image class does not depend on the position of the image object. Another regularization technique is dropout (Section 3.8). It prevents an unnecessary collaboration between model weights, serving as a way of model averaging. Such regularization techniques

are crucially important for good performance, so it is necessary to know them, understand their theoretical background, and the connections between them.

1.1 Motivation of this Work

A significant growth of interest in the area of neural networks has happened in the last few years ([59]). The research community has grown exponentially, involving a number of people who are not familiar enough with the results achieved in previous years. Some of the ideas are rediscovered, and get a new name. Since the problem of regularization is one of the most important, this also happens with the articles describing various regularization techniques. However, the most recent overview of regularization methods for neural networks was published about 10 years ago [8]. One of the goals of this thesis was to provide an overview of the most common regularization methods, and describe some of the others from the regularization point of view.

It often happens that some of the ideas are proposed without a strong theoretical basis, being based on intuition and confirmed by experiments. In fact, there is a substantial lack of knowledge about the theoretical properties of neural networks. These questions are regularly posed by the leading researchers in the area [58]. For some ideas the theoretical background comes soon after they were presented, for others this question remains open for a long time. Sometimes it happens that the same algorithm can be proposed from different points of view. In this case the analysis of the reasons helps to establish hidden connections between parameters and better understand their meaning. Another goal of the thesis was to describe the theoretical background for regularization methods and show the connections whenever it is possible.

Deep neural networks (DNN) are typically used when the classification function is complex enough (2.3.3), and simpler models are not able to achieve a required accuracy. Usually such classification tasks have a large number of features. This is why most of the current applications of DNN are in the areas of video, audio and text processing. Notice, that besides feature values the objects in these areas contain additional information, which describes feature **location**.

For example, image pixels additionally have two coordinates in X and Y axes. Features of various time series contain a value of time when this feature was taken. This is important information for a classifier, which has to be taken into account, and incorporated in the learning process. Some of the regularizers are particularly targeted for this goal (Sections 3.4, 3.5), restricting a classifier from learning an inappropriate model. In this thesis we aim to pay particular attention to such regularizers.

Since the area of neural networks has suffered from a lack of attention in previous years ([59]), some of quite simple but efficient regularization methods have not been discovered till now. Thus, we wanted to fill in this gap, and present the results of our research. However, theory always has to be confirmed by practice. The last goal was to consider one of the challenging problems from the real world and evaluate the theory on this problem. The discussion of proposed topics and the results of experiments are presented in the next chapters.

1.2 Research Contributions

Each chapter of this thesis addresses a particular aspect of the previously described goals. Here we give a short overview of each of them.

In Chapter 3 we present a comprehensive overview of the most widely used regularization methods, such as L_1 and L_2-norm weight regularization, early stopping, convolutional and subsampling layers, dropout, and provide their theoretical explanation. We demonstrate a connection between different methods and their parameters whenever it is possible. In particular, we show that most of them can be viewed as a form of a standard L_2-norm regularizer, which prevents the weights from becoming too large. In the end we describe two methods (Tangent backpropagation and Adversarial Training), which aim to improve the robustness of neural networks.

In Chapter 4 we propose an Invariant Backpropagation algorithm (IBP). Training of neural networks is performed using the standard backpropagation algorithm with two passes: forward and backward. On the forward pass it computes the predictions for the current input and the loss function, while on

the backward pass it computes the derivatives of the loss function with respect to the input and parameters. The derivatives for parameters are used for their updates, while the derivatives for the input are not used. At the same time, the nature of the data usually implies that two very close data points have the same label. This means that the predictions of a classifier should not change quickly near the points of a dataset, i.e., their derivatives should be close to zero. To enforce this, we propose a natural extension of the backpropagation algorithm, that minimizes the length of the vector of derivatives of the loss function with respect to the input values together with the minimization of the loss function itself. This extension requires an additional forward pass performed on these derivatives (not used in standard BP), so the IBP is just 50% slower than the standard BP. We apply our algorithm to a collection of datasets for image classification, confirm its theoretically established properties and demonstrate an improvement of the classification accuracy with respect to the standard backpropagation algorithm in the majority of cases. In the end we perform a theoretical and experimental comparison with the Tangent Backpropagation algorithm and Adversarial Training, which aim to address the same goal. We show that the difference between them lies in the objective functions that they minimize. Moreover, we show that both Tangent BP and Adversarial Training can be sped up in order to achieve performance comparable with Invariant BP.

In Chapter 5 we study the problem of searching the best values of hyperparameters that determine the model complexity (Section 2.3.3). For example, the proposed Invariant Backpropagation algorithm requires an additional hyperparameter that defines the strength of regularization, and therefore affects the flexibility of a classifier. In order to achieve the best results, the initial value of this parameter needs to be carefully chosen. Usually hyperparameters are chosen using a validation set or cross-validation. However, these methods might not be accurate and can be slow. We propose a new method which can estimate the classifier performance using only the training set, and evaluate it on Support Vector Machines with RBF kernels. Our method exploits the well-known Akaike and Bayesian Information Criteria, which measure the balance between the quality of representation and the amount of information required for it. We present two alternative approaches for calculating the likelihood functions for

these formulas. The first approach uses the SVM objective function, which is computationally free. The second approach directly estimates the probability that the SVM hyperplane is the optimal classifier for the given class probability distribution by analyzing the disposition of points in the kernel feature space. We experimentally compare our approaches with several existing methods and show they are able to predict the best values of hyperparameters with sufficient accuracy, whilst also having low running time compared with cross-validation. In the end we provide clues how the described algorithm can be applied for training a neural network structure.

In Chapter 6 we investigate a practical challenging problem of deception detection from visual cues. The psychology literature indicates that people are able to detect deception with an accuracy, that is slightly better than a random guess ([9], [107], [125]). At the same time, trained individuals were shown to be able to detect liars with an accuracy of up to 73% ([24], [25]). This result confirms that visual and audio channels contain enough information to detect deception. In this thesis we describe an automated multilevel system of video processing and feature engineering based on facial movements. We demonstrate that these features provide a classification accuracy that is statistically significantly higher than a random guess. Another contribution of this thesis is the collection of one of the largest datasets of videos with truthful and deceptive people recorded in more natural conditions than others. This is one of the problems of video classification, so appropriate deep learning models can be applied. However, this task is too complicated for supervised learning. We provide an insight about possible solutions in Section 7.2.3.

In Chapter 7 we summarize the contribution of the previous chapters, and indicate possible directions of future research.

Chapter 2

Background

2.1 Machine learning

The general task of machine learning is to teach a machine to produce a desired output for a given input, similar to what the word "learning" means for people. There are three main types of machine learning:

- *supervised* learning - when a dataset of input with the provided correct output is given. The output is usually called *labels*. In this case the output for the new input samples should be similar to the labels of similar samples in the dataset. This is the most common type of machine learning.
- *unsupervised* learning - when only the input data is given. In this case the goal is to learn the underlying structure of data. It assumes learning of data probability distribution, that can generate values similar to the input data, or another data representation with desired properties.
- *reinforcement* learning - when labels (a reward) are based on input data, the machine output, and the output of another agent which also depends on the machine output. This type of learning is used to teach a machine to play games.

If a set of possible label values in a problem of supervised learning is discrete, it is called a *classification* problem. In this case the labels are often called

classes, and the algorithm of predicting a class is called a *classifier*. If a set of possible label values is continuous, it is called a *regression* problem. In this work we mostly describe the methods of supervised learning.

2.2 Neural networks

2.2.1 The beginning

The history of neural networks started in 1943, when a neurophysiologist Warren McCulloch and a mathematician Walter Pitts proposed a mathematical model [69] of neurons in the brain and demonstrated that combined together they can compute any arithmetic and logical function. They also shown that neurons can be implemented with electrical circuits. This work had a great influence. John von Neumann in his works [105, 106] supported the idea of modelling brain activity using artificial neurons.

In the book "The Organization of Behavior" [73] Donald Hebb suggested a learning law for connections between neurons in a brain - they become stronger each time they are used. He performed a set of psychological experiments that confirmed his idea. The first attempt to implement artificial neurons was made in 1951 by Marvin Minsky. This machine was called Snark. While the attempt was technically successful, it did not attract much attention [117].

2.2.2 Simple perceptron

Let us consider a model of a single neuron. Formally it is a function of an argument that is linear with respect to the input:

$$\hat{y} = f(x) = \phi(x \cdot w + b) = \phi(\sum_{i=1}^{N} x^i w^i + b)$$

Here vector x is an input vector, w is a vector of parameters also called *weights*, and b is the *bias*. The vector w and the bias b need to be tuned in order to make $\hat{y}$ close to the desired output y.

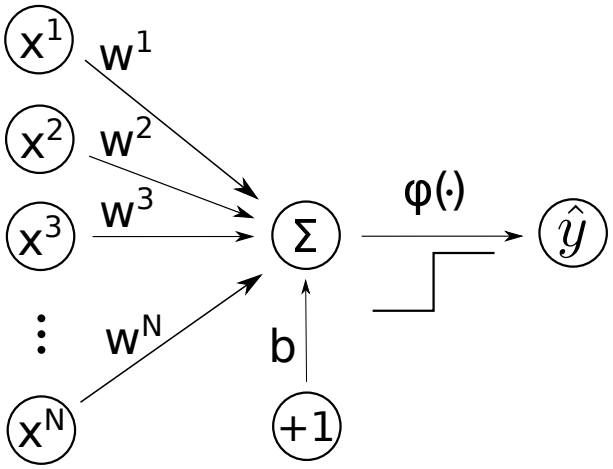


Figure 2.1: Simple perceptron.

To avoid the bias term, we can extend the input vector x by adding an additional value 1, and join w and b into a single vector of weights. Thus, now on we always consider the extended vector x with the last element equal to 1, and the extended matrix of weights w, in which the last row is the vector of biases b, unless the opposite is stated.

The function $\phi(z)$ is called an *activation* or a *transfer* function and is typically nonlinear. If $\phi(z)$ is a Heaviside step function, i.e., $\phi(z) = I(z > 0)$, where $I(z)$ is the indicator, a neuron has only two possible outputs: 0 and 1, which might correspond to two different classes.

In 1957 Frank Rosenblatt proposed an algorithm of learning the weights of a single neuron classifier. Similar to the ideas of Donald Hebb, it updates the weights such that the argument of the activation function $x \cdot w$ increases for the positive class inputs, and decreases for the negative class inputs. Formally, the rule is the following:

$$w \leftarrow w - (\hat{y} - y)x,$$

Here $y \in \{0, 1\}$ and $\hat{y} \in \{0, 1\}$ are the correct and the predicted labels. If there is no mistake, i.e., $y = \hat{y}$, the algorithm does not update weights. It has been proved that in the case of two linearly separable classes the algorithm converges in a finite number of steps. [77]. In contrast, if the classes are not separable,

it never converges. Indeed, in this case there is always a set of misclassified samples that will cause an infinite sequence of changes of weights

The model of a single neuron classifier with a Heaviside step function and the corresponding learning algorithm is called a *perceptron*.

2.2.3 Multiclass perceptron

It is possible to extend a perceptron to work with multiple classes. If the number of classes $M > 2$, the number of neurons should be also M. Each of them needs to be connected to the input vector by its own set of weights and give its own prediction. Thus, we get a vector of predictions:

$$\hat{y} : \hat{y}^i = f^i(x), \forall i = 1, \ldots, M$$

In this case the predicted class is given by the maximum value of this vector

$$k : \hat{y}^k \geq \hat{y}^i, \forall i = 1, \ldots, M$$

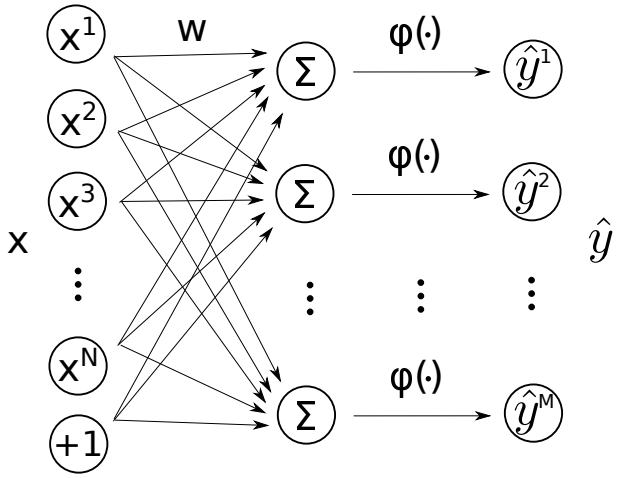


Figure 2.2: Multiclass perceptron.

The perceptron learning rule remains the same. It just need to be indepen-

dently applied to every output neuron. It can be written in a matrix form as

$$w \leftarrow w - x^T \cdot (\hat{y} - y), \tag{2.1}$$

where w is the matrix of size $N \times M$, x is a vector $1 \times N$ and $\hat{y} - y$ is the vector of errors of size $1 \times M$. Here $y_{i=1,\dots,M} \in \{0, 1\}$ is vector of labels, where 1 corresponds to the correct class. Usually the object x belongs to only one class, i.e., $\sum_{i=1}^{M} y_i = 1$. We will always use this representation in the future.

In 1957 - 1958 Frank Rosenblatt and others implemented in hardware a multiclass perceptron with a 20×20 receptive field, i.e., 400 inputs, and 8 outputs. This machine was called Mark-1. It was the first successful attempt to build a neurocomputer. It could be trained by presenting cards with different geometric objects, and could guess the objects on the newly shown cards after it.

The machine caused a huge amount of interest in the public. However, it was shown that the perceptron has its own limitations. For example, it does not converge if the classes are not linearly separable, which means that it could not implement even such a simple function as XOR.

2.2.4 Gradient descent learning rule

At the same time Bernard Widrow and his student Ted Hoff introduced a slightly modified learning rule [115], called $\mu - LMS$. Instead of fixing misclassification errors they proposed to use the squared error as a measure of quality

$$l(y, f(x)) = \frac{1}{2}||y - f(x)||_2^2 = \frac{1}{2}\sum_{i=1}^{M}(y^i - f^i(x))^2, \tag{2.2}$$

and minimize the *mean squared error* (MSE)

$$L(w) = \frac{1}{N}\sum_{i=1}^{N} l(y_i, f(x_i)), \tag{2.3}$$

over the objects of the training set. For this purpose they suggested to use a method of *gradient descent*

$$w \leftarrow w - \alpha \frac{\partial L(w)}{\partial w}, \tag{2.4}$$

where α is a learning rate, which regulated the speed of convergence. The algorithm moves the vector of weights w in the direction that decreases the value of the loss function $L(w)$. If no activation function $\phi(z)$ is used, the function $L(w)$ is a convex hyperparaboloidal surface, which achieves its minimum in a single point w^*. Therefore, with a proper choice of α the algorithm is guaranteed to converge.

In fact, it is not necessary to compute the gradient $\nabla_w L(w)$ over the whole training set. On each iteration a good approximation can be obtained by averaging over its randomly chosen subset of a smaller size, called *batch*. It increases the chances to converge to a global minima if $L(w)$ is not convex, and significantly reduces computation time. This modification is called *stochastic gradient descent* (SGD). For simplicity of further discussion we always assume that the size of the batch is 1, i.e., $L(w) = l(y, f(x))$, unless the opposite is stated.

To use SGD, it is required to compute the weight gradients $\nabla_w l(x; w)$. They can be computed using the chain rule

$$\frac{\partial L}{\partial w^{ij}} = \frac{\partial L}{\partial \hat{y}^j} \frac{\partial \hat{y}^j}{\partial w^{ij}}, \; \forall i = 1, \ldots, N, \; j = 1, \ldots, M \tag{2.5}$$

For the squared loss function $l(y, f(x))$ the derivatives with respect to predictions $\hat{y}$ are computed as

$$\frac{\partial L}{\partial \hat{y}^i} = \hat{y}^i - y^i \; \Rightarrow \; \frac{\partial L}{\partial \hat{y}} = \hat{y} - y \tag{2.6}$$

However, in order to compute $\partial \hat{y}^j / \partial w^{ij}$ the function $\phi(z)$ needs to be differentiable, so the Heaviside step function cannot be used anymore. If no function

$\phi(z)$ is used, the gradients $\nabla_w \hat{y}$ are simply the elements of the input vector

$$\frac{\partial \hat{y}^j}{\partial w^{ij}} = x^i, \forall i = 1, \ldots, N, \ \ \forall j = 1, \ldots, M$$

Thus we can write the μ-LMS rule in the matrix form as

$$w \leftarrow w - \alpha \frac{\partial L}{\partial w} = w - \alpha x^T \cdot (y - \hat{y})$$

We thus see that this is the same rule as Equation (2.1), except that the step size can now be tuned by the hyperparameter α. In [116] it was shown that if the input vector x is normalized to have a unit lenght, the choice of $0 < \alpha < 2$ guarantees the convergence of the algorithm. This learning rule was called α-LMS. A single neuron with the α-LMS learning rule was called *Adaline* (ADAptive LINear Element). A model with several output neurons was called *Madaline* (Many Adaline). Several Madaline machines were built in hardware and demonstrated success on a number of toy problems [117].

As it follows from the definition (2.2), the gradient descent learning rule (Equation(2.4)) can be applied to continuous targets y, i.e., to be also used for the regression problem.

If some nontrivial function $\phi(z)$ is used, the learning rule also contains its gradient:

$$w \leftarrow w - \alpha x^T \cdot \nabla_z \phi(z) \cdot (\hat{y} - y),$$

This was called δ-LMS learning rule. We discuss nonlinear activation functions in Section 2.2.8.

2.2.5 Multilayer perceptron

While perceptrons implement a linear function in the input vector space, in real applications the separating boundary between classes is rarely linear. However, it might be possible that classes are linearly separable in some other space. In this case an additional transformation is required.

In the multiclass perceptron the vector of predictions $\hat{y}$ is a linear function of

the input vector x and can be written in a matrix form as

$$\hat{y} = x \cdot w$$

Here x and y can be considered as two layers, *fully connected* by the weights in the matrix w. The additional transformation can be performed by another layer of neurons located between the input and output, and fully connected with both of them. It is called a *hidden* layer.

If we enumerate the layers as $y_0 = x$, y_1 and $y_2 = \hat{y}$, we can write that

$$y_1 = y_0 \cdot w_1,\ y_2 = y_1 \cdot w_2$$

However, such a perceptron is equivalent to a perceptron without a hidden layer with the matrix of weights $w = w_1 \cdot w_2$. In order to make a hidden layer useful, the transformation function of its neurons has to be nonlinear.

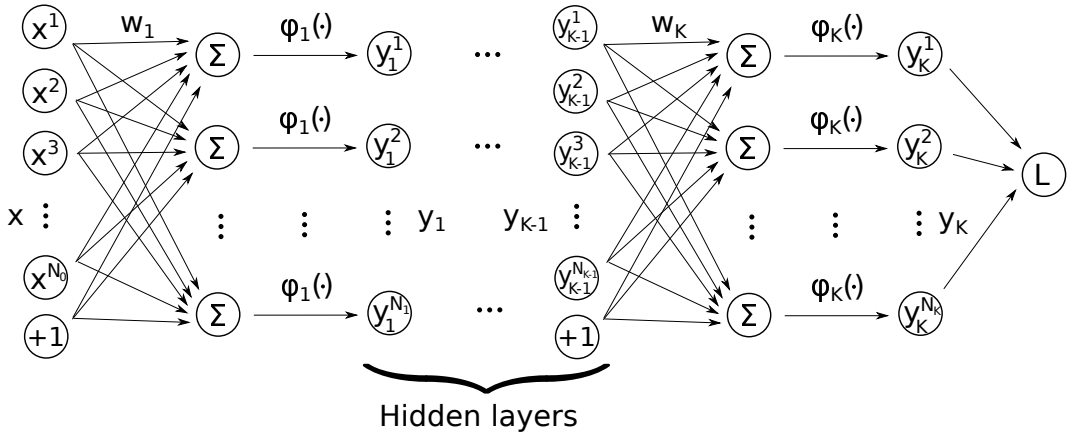


Figure 2.3: Multilayer perceptron.

In fact, the perceptron of Rosenblatt as well as Madaline machines had a hidden layer of neurons, but the weights of connections with the input layer were predefined. Since these machines were built in hardware, it was easier to implement a Heaviside step activation function $\phi(z) = I(z > 0)$. In order to be able to tune the weights of the hidden layer as well, in 1962 there was proposed Adaline Rule I [114]. It was the first popular algorithm of training a multilayer

perceptron.

Multilayer perceptrons contain only a particular type of layers, called fully connected layers, that have an independent connection between each pair of neurons. Generalized perceptrons that might contain layers of other types are called *neural networks*.

2.2.6 Backpropagation algorithm

Applying the ideas of Bryson [49] and Kelley[10] to multilayer perceptrons, Paul Werbos [112] proposed an extension of the $\delta - LMS$ learning rule, which was called the *backpropagation algorithm*.

Let us consider a multilayer perceptron with K layers of neurons: $y_0 = x$ input vector of size $1 \times N$, and the following K layers

$$y_k = \phi_k(y_{k-1} \cdot w_k),\ \forall k = 1, \ldots, K$$

with $\phi_k(z)$ activation functions, where y_K is the output vector of the size $1 \times M$. If the functions $\phi_k(z)$ are differentiable, we can compute the gradients for layer y_{k-1} using the gradients for the layer y_k using the chain rule as follows:

$$\frac{\partial L}{\partial y^i_{k-1}} = \sum_{j=1}^{M} \frac{\partial L}{\partial y^j_k} \cdot \frac{\partial y^j_k}{\partial y^i_{k-1}} = \sum_{j=1}^{M} \frac{\partial L}{\partial y^j_k} \cdot \left.\frac{\partial \phi_k(z)}{\partial z}\right|_{z=y_{k-1}w^j_k} \cdot w^{ij}_k \tag{2.7}$$

Here w^j_k is the vector of weights, connecting the layer $k-1$ with the neuron j on the layer k. As we can see, the previous layer's gradients $\partial L/\partial y_{k-1}$ depend only on the current layer's gradients $\partial L/\partial y_k$, the activation function derivative $\nabla_z \phi_k(z)$ and the layer weights w_k. Therefore, it is possible to iteratively compute the gradients for all layers from $K-1$ to 0, using $\partial L/\partial y_K$ (2.6) on the first iteration. The weight gradients $\partial L/\partial w_k$ can also be computed using the chain rule

$$\frac{\partial L}{\partial w^{ij}_k} = \frac{\partial L}{\partial y^j_k} \cdot \frac{\partial y^j_k}{\partial w^{ij}_k} = \frac{\partial L}{\partial y^j_k} \cdot \left.\frac{\partial \phi_k(z)}{\partial z}\right|_{z=y_{k-1}w^j_k} \cdot y^i_{k-1}, \tag{2.8}$$

and therefore can be obtained iteratively for all layers from K to 1 once the

gradients $\partial L/\partial y_k$ become available.

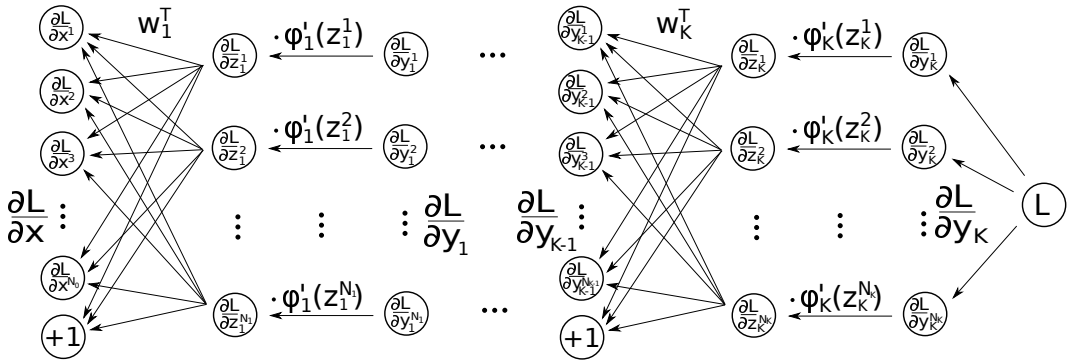


Figure 2.4: The backward pass of the backpropagation algorithm. The forward pass is represented in Fig. 2.3

The algorithm can be split into three parts: *forward* pass, *backward* pass, and weight gradient computation. On the forward pass we initialize the input vector y_0 with some training object x and iteratively compute the following layer values y_i from 1 to K. On the backward pass we initialize prediction gradients $\partial L/\partial y_K$ using Equation (2.6), and propagate them back through the network for the layers from $K-1$ to 0, obtaining the gradients $\partial L/\partial y_i$. The weight gradients $\partial L/\partial w_i$ can then be computed in parallel or after the backward pass, using the layer values y_i and their gradients $\partial L/\partial y_i$. The weight gradients are then used to update the weights using Equation (2.4). When the size of a batch is more than 1, the mean of the weight gradients is used. Usually the backpropagation algorithm processes the same training set many times, until the test error stops to decrease. Each processing iteration is called *epoch*.

Apart the method of stochastic gradient descent, there exist a number of other methods of searching the minimum of the loss function. The most popular of them are AdaGrad [21], which allows to efficiently adapt the learning rate, Nesterov accelerated gradient descent [75] with the convergence rate $1/t^2$, LBFGS algorithm [64], which is based on the approximated second order gradients, and others. However, the SGD remains the most commonly used method of training neural networks.

2.2.7 Matrix representation

It is convenient to consider the activation functions as a separate transformation layer, and use matrix notation. In this case we can formalize a layer as a function

$$y_i = f_i(y_{i-1}, w_i), \tag{2.9}$$

which transforms a vector y_{i-1} to a vector y_i. Moreover, the loss function L can also be considered as the last layer y_{K+1} of the length 1. The forward pass is thus a composition of functions $l(x) = f_{K+1}(f_K(\dots f_1(x)\dots))$, applied to the input vector x.

Let us denote the vectors of derivatives with respect to layer values $\partial L/\partial y_i$ as dy_i. Then, similar to the forward propagating functions $y_i = f_i(y_{i-1}, w_i)$, we can define backward propagating functions $dy_{i-1} = \tilde{f}_i(dy_i, w_i)$. We refer to them as *reverse* functions. The first part of Equation (2.7) can be rewritten in matrix form as

$$dy_{i-1} = \tilde{f}_i(dy_i, w_i) = dy_i \cdot J_i^y(y_{i-1}), \tag{2.10}$$

where $J_i^y(y_{i-1})$ is the Jacobian matrix of the derivatives $\partial y_i^j/\partial y_{i-1}^k$. The backward pass is thus a consecutive matrix multiplication of the Jacobians $J_i^y(y_{i-1})_{i=K,\dots,1}$ of layer functions $f_i(y_{i-1}, w_i)$, computed at the points y_{i-1}. Note, that the first Jacobian $J_{K+1}(y_K)$ is the vector of derivatives $dy_K = \partial L/\partial y_K$ of the loss function L with respect to predictions y_K.

Most of the functions f_i are one of two types:

- **linear, with weights**, i.e., $y_i = y_{i-1} \cdot w_i \;\Rightarrow\; dy_{i-1} = dy_i \cdot w_i^T$.
- **nonlinear, without weights**, i.e., $y_i = \phi(y_{i-1}) \;\Rightarrow\; dy_{i-1} = dy_i \cdot J_\phi(y_{i-1})$

Usually the functions $\phi(x)$ are element-wise, so they have a squared and diagonal Jacobian $J_\phi(y_{i-1})$.

Next, let us denote the vector of weight gradients $\partial L/\partial w_i$ as dw_i. Then we can rewrite Equation (2.5) in matrix form as $dw_i = J_i^w(y_{i-1}) \cdot dy_i$, where $J_i^w(y_{i-1})$ is the Jacobian matrix of the derivatives with respect to weights $\partial y_i^j/\partial w_i^{kl}$. However, if f_i is a linear function, the Jacobian $J_i^w(y_{i-1})$ is equivalent to the vector

y_{i-1}^T, so

$$dw_i = y_{i-1}^T \cdot dy_i \tag{2.11}$$

Nonlinear functions usually do not contain weights, so there is no need to compute their weight gradients.

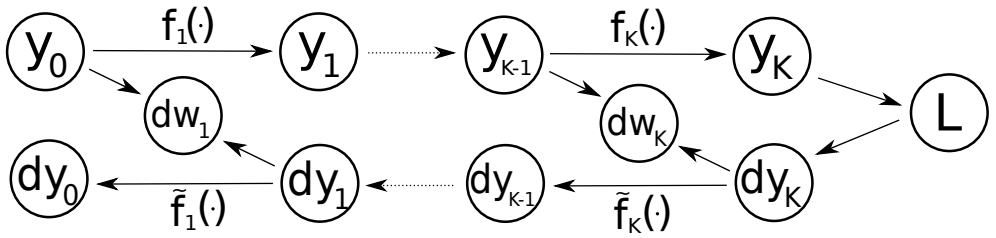


Figure 2.5: The scheme of the backpropagation algorithm in matrix form.

2.2.8 Nonlinear functions

The backpropagation algorithm requires all activation functions to be differentiable. For this reason the Heaviside step function cannot be used anymore. The easiest way is to approximate it with another differentiable function, which has a similar shape. Such functions are called *sigmoid* functions. The two most used forms of them are the *logistic* function

$$\phi(z) = sigm(z) = \frac{1}{1+e^{-z}}, \Rightarrow \forall z\, \phi(z) \in (0,1)$$

and hyperbolic tangent

$$\phi(z) = tanh(z) = \frac{1-e^{-2z}}{1+e^{-2z}}, \Rightarrow \forall z\, \phi(z) \in (-1,1)$$

Moreover, these functions have convenient gradients $\nabla_z \phi(z)$. For the logistic function

$$\frac{\partial \phi(z)}{\partial z} = \frac{e^{-z}}{(1+e^{-z})^2} = \phi(z)(1-\phi(z)),$$

and for the hyperbolic tangent

$$\frac{\partial \phi(z)}{\partial z} = \frac{4e^{-2z}}{(1+e^{-2z})^2} = 1 - \phi^2(z)$$

The derivatives of these functions depend only on the neuron value itself, so their Jacobian matrix in Equation (2.10) is diagonal.

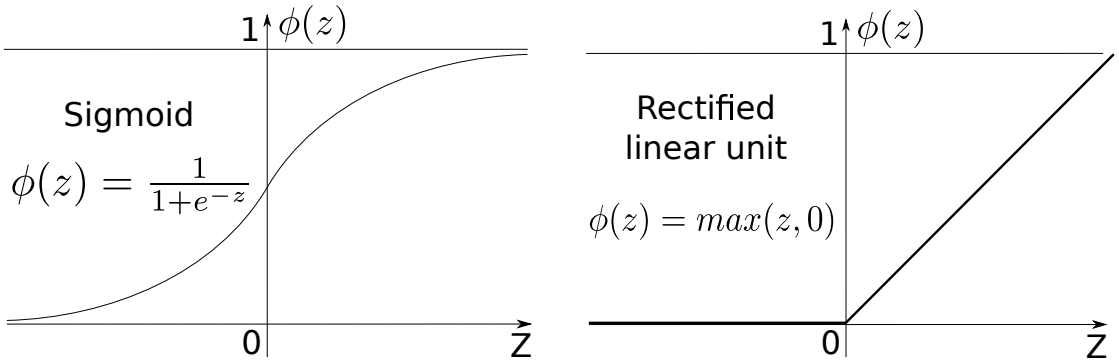
Recently it was shown that sigmoid functions $\phi(z)$ are not the best choice. They have an inconvenient property: once $\phi(z)$ is too close to 0 or 1, its derivative become too close to 0, so it requires many iterations to train the network. At the same time, a simple function such as the *REctified Linear Unit* (RELU) [74]

$$\phi(z) = max(z, 0) = zI(z > 0)$$

is much faster for computation and does not have this problem. It was shown [54] that it allows a network to be trained several times faster than the standard logistic function. Moreover, its derivative is similar to the function itself

$$\frac{\partial \phi(z)}{\partial z} = I(z > 0) \Rightarrow f_i(z) = \tilde{f}_i(z)$$

so the propagation on the forward and backward passes can be implemented by the same function.



(a) Logistic sigmoid function.

(b) Rectified linear unit (ReLU).

Figure 2.6: The shape of the most common nonlinear activation functions.

2.2.9 Adaptation for classification problem

In the case of classification problems it is desirable to make the predictions $\hat{y}$ output the probabilities of being a particular class, so they satisfy two requirements

$$\hat{y}^i \geq 0, \sum_{i=1}^{M} \hat{y}^i = 1$$

Sigmoid function can guarantee the first constraint, but not the second one. In this case it is convenient to use the *softmax* activation function

$$\phi^i(z) = \frac{e^{z^i}}{\sum_{j=1}^{M} e^{z^j}} \Rightarrow \sum_{i=1}^{M} \phi^i(z) = 1 \tag{2.12}$$

Thus, it depends on all layer values, not only the value of a single neuron as other nonlinear functions. The softmax function also has a convenient derivative

$$\frac{\partial \phi^i(z)}{\partial z^j} = \phi^i(z)(\delta_{ij} - \phi^j(z))),$$

where $\delta_{ij} = I(i = j)$ is the Kronecker delta function. Note, that the Jacobian (Equation (2.10)) of the softmax function is not diagonal anymore, but is still symmetric.

However, the softmax output layer has another problem: even when a perceptron has no hidden layers, the squared loss function (2.3) is not convex with respect to w anymore. It makes SGD algorithm more difficult to find a global minima. For this reason there was proposed a *cross-entropy* loss, or *negative log-likelihood* loss, which is convex with respect to w:

$$l(y, f(x)) = -\langle y, \log f(x) \rangle = -\sum_{i=1}^{M} y^i \log f^i(x) \tag{2.13}$$

The negative logarithm is not bounded from below, but since the predictions $f^i(x)$ are restricted to be within $(0, 1)$ by the softmax layer, the value of the loss function is always positive. The use of the softmax - cross-entropy loss combination has another convenient property. Let us compute the derivatives

of $l(z) = \langle y, \log \phi(z) \rangle$ with respect to the softmax input values z:

$$\begin{aligned} \frac{\partial l(z)}{\partial z^j} &= \sum_{i=1}^{M} \frac{\partial l(z)}{\partial \phi^i(z)} \cdot \frac{\partial \phi^i(z)}{\partial z^j} = \sum_{i=1}^{M} \frac{y^i}{\phi^i(z)} \phi^i(z)(\delta_{ij} - \phi^j(z)) = \\ &= \sum_{i=1}^{M} y^i \delta_{ij} - \phi^j(z) \sum_{i=1}^{M} y^i = y^j - \phi^j(z) \Rightarrow \frac{\partial l(z)}{\partial z} = y - \phi(z) \end{aligned} \tag{2.14}$$

Therefore, the softmax input gradients have the same simple formula (Equation (2.6)) as the prediction gradients of the squared loss function (Equation (2.2)). It allows us to simplify implementation and improves numerical stability.

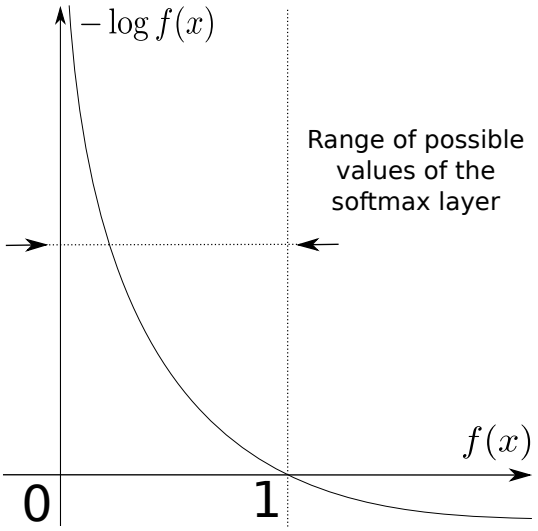


Figure 2.7: Plot of the negative logarithm function. If the input is a probability $0 \leq f(x) \leq 1$, the function is always positive.

2.2.10 Universal approximation theorem

Multilayer perceptrons have an important theoretical property. It was proven by George Cybenko in 1989 ([15]), that under some mild conditions they are able to uniformly approximate any continuous function with just one hidden layer. This result is known as the *universal approximation theorem*. In 1991 it was shown ([41]) that the theorem is true for all continuous, bounded and non-constant activation functions. This result also generalizes other theorems about uniform

representation: for polynomial functions (theorem of Stone-Weierstrass), for radial basis functions, Fourier series, and others.

Here we provide a proof sketch. Let us consider some continuous function $f(x)$ in a unit cube $[0,1]$, and a multilayer perceptron with sigmoid transfer functions. The proof consists of two statements:

1. A continuous function on a compact space can be approximated by a piecewise constant function.

2. A piecewise function can be implemented in the following way:

    - The neurons of a hidden layer implement an indicator function of all cube regions. Sigmoid functions are used as approximators.
    - The final layer neuron is a sum of the indicator functions of all regions with appropriate weights.

Thus, depending on the required approximation precision ϵ we choose a required number of regions (neurons) such that on each of them the variation of $f(x)$ is less than ϵ. An even more strong result about exact representation was proved by Andrey Kolmogorov in 1957 [51]. However, in this case the activation functions are not necessarily the same. These results ensure that there are no theoretical limitations on the performance of multilayer perceptrons.

2.2.11 Recurrent neural networks

A special type of neural networks is a *recurrent* neural network (RNN). It is trained on the ordered **sequences** of input values, and can predict the next element of a sequence. Unlike feedforward neural networks, which might observe all input simultaneously, it observes only one input value at a time. The information about previous values is stored in the hidden layers, which are used to calculate the hidden layer values for the current input.

RNN might be considered as an infinite sequence of replications of a feedforward neural network, connected with each other. Let us denote the input vector at a time t as x^t. Using the similar notation as earlier, we will also refer

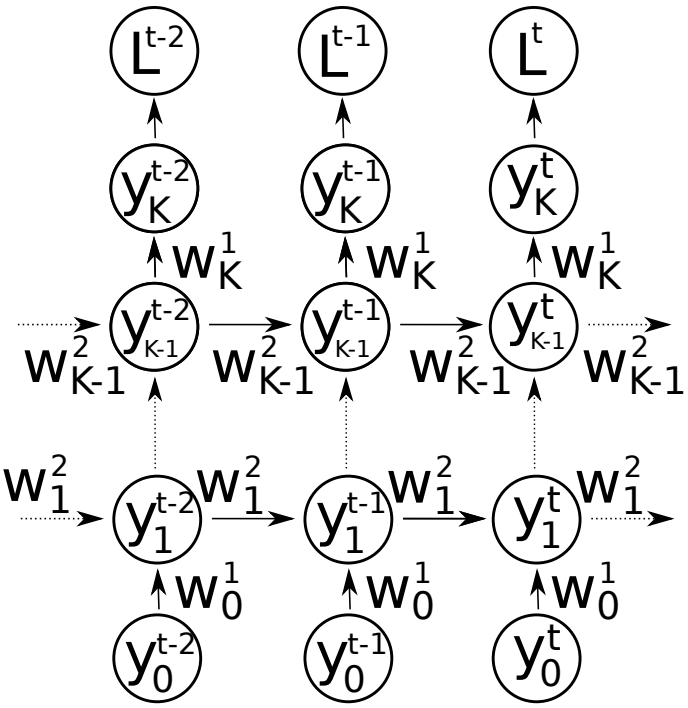


Figure 2.8: Visualization of a recurrent neural network

to it as y_0^t. A recurrent neural network can be formalized with the following formula:

$$y_i^t = \phi(y_{i-1}^t \cdot w_i^1 + y_i^{t-1} \cdot w_i^2), \;\; i = 0, \ldots, K, \;\; t \geq 0$$

In other words, the current layer values depend on the previous layer values at a time t and a current layer values at a time $t-1$. However, the dimensions i and t are not equivalent. While the range of i is defined by the network structure, the range of t is limited only by the length of the input sequence. As earlier, it is convenient to consider the activation function $\phi(z)$ as a separate layer, which does not depend on parameters w.

RNN training is performed using a modification of the backpropagation algorithm called *Backpropagation through time* (BPTT) [113]. Its pseudo-code is provided in 2.1. However, the standard RNN is suffering from the problem of vanishing gradients [39]. To avoid it, a modification of RNN called *Long Short-Term Memory* (LSTM) network [40] is usually used.

Algorithm 2.1 Backpropagation through time

Assign all gradients to 0
for $t = T$ **to** $T - S$ **do**
    for $i = K$ **to** 1 **do**
        $dy_{i-1}^t \leftarrow dy_{i-1}^t + dy_i^t \cdot (\partial y_i^t / \partial y_{i-1}^t)$
        if $(\partial y_i^t / \partial y_{i-1}^t)$ is w_i^1 **do**
            $dw_i^1 \leftarrow dw_i^1 + (y_{i-1}^t)^T \cdot dy_i^t$
        if $(\partial y_i^t / \partial y_i^{t-1})$ is w_i^2 **do**
            $dw_i^2 \leftarrow dw_i^2 + (y_i^{t-1})^T \cdot dy_i^t$
            $dy_i^{t-1} \leftarrow dy_i^t \cdot (w_i^2)^T$
for $i = K$ **to** 1 **do**
    if $(\partial y_i^t / \partial y_{i-1}^t)$ is w_i^1 **do**
        $dw_i^1 \leftarrow dw_i^1 / S$
    if $(\partial y_i^t / \partial y_i^{t-1})$ is w_i^2 **do**
        $dw_i^2 \leftarrow dw_i^2 / S$

2.3 Overfitting and regularization

The goal of any supervised task of machine learning is to build a machine such that being fed by similar objects to the training set, it outputs similar predictions. Thus, the machine always depends on the training set. However, if the machine is too complex, it might depend on it too much. In this section we discuss this phenomenon and the ways to avoid it.

2.3.1 Regression problem

Let us consider the regression problem, i.e., the problem of constructing a function $f(x)$ based on a *training set* $D = \{(x_1, y_1), \ldots, (x_N, y_N)\}$, that would approximate the values y for the future observations x.

In order to approximate y, the function $f(x)$ needs to have a set of parameters w that can be tuned based on the training set D, so we can denote $f(x)$ as $f(x; w)$. The process of tuning the parameters w is called *training*. Typically it

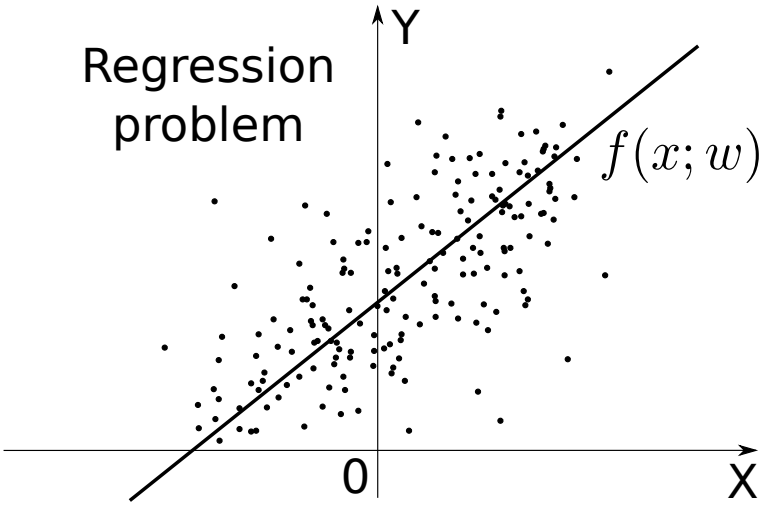


Figure 2.9: Visualization of a linear regression

assumes the minimization of some function $L_{min}(f(x;w))$ with respect to w, i.e.,

$$w^* = \arg\min_w L_{min}(f(x;w))$$

Let the loss function $l(y, f(x))$ be a measure of approximation at a particular point (x, y). A common choice for it is the mean squared error (MSE):

$$l(y, f(x)) = ||y - f(x)||_2^2 \tag{2.15}$$

Then for a given training set D_N of size N we can define the *training error* as a mean error over the training set

$$L_{train} = \frac{1}{N}\sum_{i=1}^{N} l(y_i, f(x_i)) = \frac{1}{N}\sum_{i=1}^{N}(y_i - f(x_i))^2,\ (x_i, y_i) \in D_N \tag{2.16}$$

For simplicity we consider only the MSE function and one-dimensional vector y, but this is not necessary. Since the function $f(x)$ needs to have low error on the training set D, the most common choice is $L_{min}(f(x;w)) = L_{train}(f(x;w))$, i.e., the goal of training is to find a set w that minimizes L_{train}.

2.3.2 Bias-variance tradeoff

The first comprehensive description of the model bias-variance tradeoff phenomenon was given in [31].

Let us assume that all pairs (x, y) are sampled from some joint distribution $P(x, y) = P(x)P(y|x)$. Then, for a given x, the average error $l(y, f(x))$ is

$$L_x = E_{P(y|x)} l(y, f(x)) = E_{P(y|x)}[(y - f(x))^2|x]$$

We can split L_x on the variance of y and the regression error:

$$\begin{aligned} L_x &= E[(y - E[y|x] + E[y|x] - f(x))^2|x] = \\ &= E[(y - E[y|x])^2|x] + E[(E[y|x] - f(x))^2|x] + \\ &+ 2E[(y - E[y|x])|x] \cdot E[(E[y|x] - f(x))|x] = \\ &= E[(y - E[y|x])^2|x] + (E[y|x] - f(x))^2 = \\ &= Var(y|x) + L_{reg}(f(x)) \end{aligned} \tag{2.17}$$

The variance $Var(y|x)$ depends only on the distribution P and therefore is irreducible. Notice, that the best solution for $f(x)$ that minimizes L is the expected value of y, i.e., $f(x) = E_{P(y|x)}[y|x]$.

Since $f(x)$ is based on D, we refer to it as $f(x; D)$. We can evaluate the performance of $f(x; D)$ for a given D using $L_{reg}(f(x))$, but we are actually interested in how good it is on average, i.e., what is

$$E_D[L_{reg}(f(x; D))] = E_D[(f(x; D) - E[y|x])^2]$$

Similar to (2.17), for any x, it can be decomposed into two parts:

$$\begin{aligned} E_D[L_{reg}(f(x; D))] &= E_D[(f(x; D) - E_D[f(x; D)] + E_D[f(x; D)] - E[y|x])^2] = \\ &= E_D[(f(x; D) - E_D[f(x; D)])^2] + E_D[(E_D[f(x; D)] - E[y|x])^2] + \\ &+ 2E_D[f(x; D) - E_D[f(x; D)]] \cdot E_D[E_D[f(x; D)] - E[y|x]] = \\ &= E_D[(f(x; D) - E_D[f(x; D)])^2] + (E_D[f(x; D)] - E[y|x])^2 = \\ &= Var(f(x; D)) + Bias(f(x; D))^2 \end{aligned} \tag{2.18}$$

The model bias shows how far on average the prediction $f(x; D)$ is from the best value $E[y|x]$. The model variance reflects how sensitive is the prediction to the training set D. Note, that both bias and variance also depend on a particular distribution P, to which the model is applied.

In many regression models there is a parameter that controls their flexibility (or complexity). For example, the flexibility of neural networks depends on the number of hidden layers and the number of neurons in it. The flexibility of the k-nearest neighbor classifier depends on the number k. The more flexible the model is, the larger is the space of possible functions $f(x; D)$. Such models have lower bias, but higher variance. Hence, there is a problem of finding the optimal value of this parameter, which yields the optimal balance between bias and variance. This problem is called the *bias-variance tradeoff*.

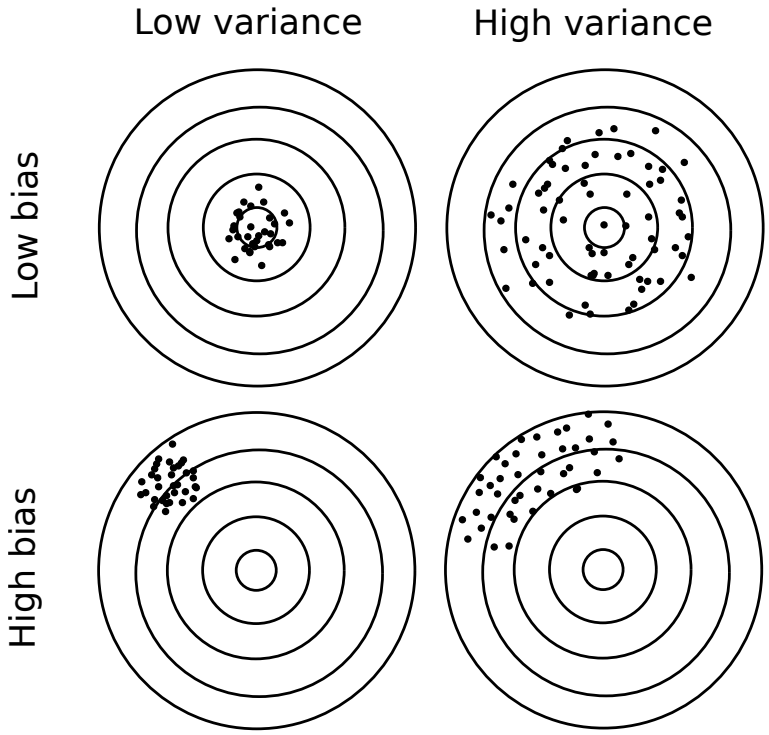


Figure 2.10: The representation of different bias-variance scenarios. The circles represent the levels of the loss function L_{reg}. Smaller circles have lower value of L_{reg}. Each dot corresponds to a different training set D. High bias prevents models from having low error. High variance does not guarantee low error.

2.3.3 VC-dimension

In Section 2.3.2 we mentioned that different degrees of models have different flexibility, which affects the balance between bias and variance. In this section we give one of the possible definitions of model flexibility, or *complexity*. One of the most common definitions is called *VC-dimension*, proposed by Vladimir Vapnik and Alexey Chervonenkis in [103].

Let us consider a classification function $f(x;w)$, where w is a set of tuned parameters. The function $f(x;w)$ *shatters* a set of points $D_N = (x_1, \ldots, x_N)$, if for any combination of labels $(y_1, \ldots, y_N)$, $y_i \in \{0, 1\}$ there exist a vector w such that the training error (Equation (2.16)) $L_{train}(f(x;w)) = 0$. In other words, the classifier $f(x;w)$ can correctly classify all samples in the training set. Then, the VC-dimension of the function $f(x;w)$ is the maximum number N, such that there exist a set of points $D_N = (x_1, \ldots, x_N)$, that can be shattered by the function $f(x;w)$.

Let us consider a linear classifier (simple perceptron) in the space of input vectors of size K. It is easy to show that its VC-dimension is $K + 1$. Indeed, for any $K + 1$ points and any $K + 1$ distances d with positive and negative signs we can find a vector of weights w of the length $K + 1$ (K for coordinates and 1 for bias) as a solution of the equation

$$X \cdot w = D \;\Rightarrow\; w = X^{-1} \cdot D$$

It is possible, because X is a squared matrix, that might be chosen such that it has non-zero determinant $|X|$, so there exists an inverse matrix X^{-1}. For $N > K + 1$ it is already not possible, what immediately follows from Radon's theorem [100].

VC-dimension allows us to estimate a test error (2.20) using a training error (2.16) and a penalty term with VC-dimension H in the following way [101]: for any $0 \leq \theta \leq 1$

$$P\left(L_{test}(f) \leq L_{train}(f) + \sqrt{\frac{H\log(2N/H) + H - \log(\theta/4)}{N}}\right) < 1 - \theta,$$

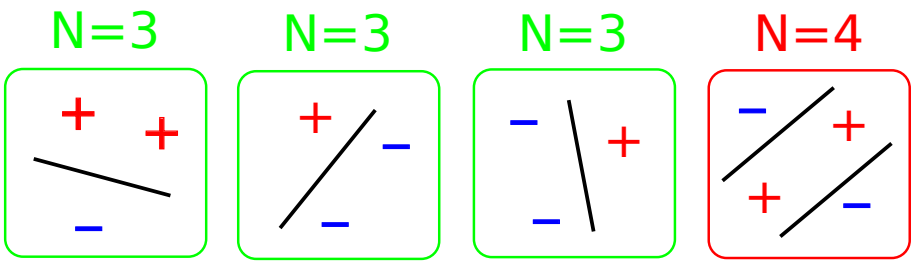


Figure 2.11: A linear classifier in a 2-dimensional space can shatter 3 points, but not 4. Therefore, its VC-dimension is 3.

where N is the size of the training set.

Usually the model complexity H depends on one (or many) hyperparameters M. The expression above sets the upper bound of L_{test}, so M can be chosen such that it minimizes this upper bound without *cross-validation* (Section 2.4.1). This method is called *structural risk minimization*. However, in practice the estimations of L_{test} obtained in this way are too large.

2.3.4 Underfitting and overfitting

The derivation of Equation (2.18) is made for a particular point x. In fact, we are more interested in the integrated loss function L_{int} over all pairs (x, y), i.e.,

$$L_{int} = E_{P(x,y)}[E_D[l(y, f(x; D))]]$$

According to Equations (2.17) and (2.18), $L_{int}(f)$ can be written as

$$\begin{aligned} L_{int}(f) &= E_{P(x,y)}[Bias(f(x;D))^2 + Var(f(x;D)) + E_D[Var(y|x)]] = \\ &= E_{P(x)}[Bias(f(x;D))^2] + E_{P(x)}[Var(f(x;D))] + E_{P(x)}[Var(y|x)] \end{aligned} \tag{2.19}$$

According to the law of large numbers (LLN), it can be approximated by averaging the values of $l(y, f(x))$ over different training sets D and a set of pairs (x, y) independently sampled from the distribution P, that are called *test*

sets. With the size S, we denote them as T_S. Thus, we can approximate L_{int} as

$$L_{test} = \frac{1}{SJ}\sum_{i=1}^{S}\sum_{j=1}^{J} l(y_i, f_j(x_i)) = \frac{1}{SJ}\sum_{i=1}^{S}\sum_{j=1}^{J}(y_i - f_j(x_i))^2,\ (x_i, y_i) \in T_S \qquad (2.20)$$

This approximation is called the *test error*. Note, that since $f(x; D_N)$ depends on the training set D_N, L_{train} (2.16) is a not an unbiased estimation of L_{int}.

Let us define a class of functions F_H of the same complexity H. This class can be given by a set of hyperparameters M. As an example we can consider a multilayer perceptron with several hidden layers, where the vector M denotes the number of neurons in each of them. Another parameter is the size of the training set N. Since the goal of training is to minimize the training error, we can define the best function $f_H(x; N, J)$ within a class F_H as

$$f_H(x; N, J) = \arg\min_{f \in F_H} L_{train}(f) \qquad (2.21)$$

Now let us consider the sequence of functions $f_H(x; N, J)$ for constant N and J and $H = 0, \ldots, \infty$. Each of them has its own test errors (2.19), approximated by Equation (2.20). The third term in Equation (2.19) does not depend on $f_H(x)$ and therefore is a constant with respect to H, so we need to examine the first two terms. In the case of $H = 0$ the function $f(x)$ is constant, so for any non-trivial distribution $P(x, y)$ it has some positive bias and zero variance. In the case of $H = \infty$ the function $f(x)$ perfectly fits any number of training samples D_N, so it has zero bias and some positive variance. Between these two states the functions of bias and variance are typically monotonic. Thus, the function $L_{int}(f)$ is bounded from below and reaches its minimum at some point H_{opt}.

The fully trained functions $f_H(x; N, J)$ with $H > H_{opt}$ are called *overfitted*, while the functions with $H < H_{opt}$ are called *underfitted*. Usually the functions with $H > H_{opt}$, which are slightly undertrained, achieve lower $L_{test}(f)$ than the fully trained functions. This is one of the ways to prevent overfitting. Notice that the optimal value H_{opt} depends not only on the distribution $P(x, y)$ and the model $f_H(x; N, J)$, but also on the size of the training set N. The larger is N, the more flexible model can be the optimal one.

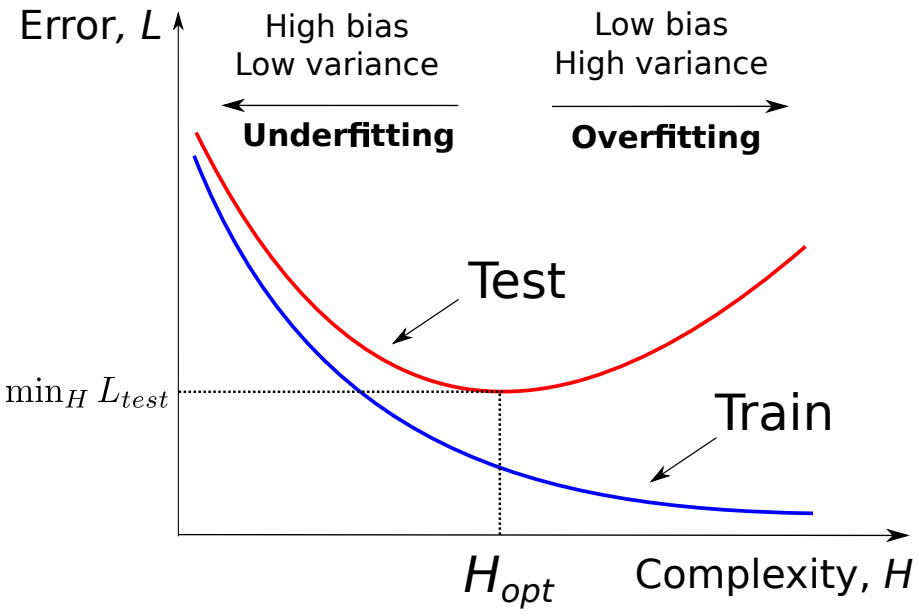


Figure 2.12: The typical shape of the train and test errors depending on the model complexity. Test error reaches its minimum at some point H_{opt}.

2.3.5 Regularization

In Section 2.3.2 we demonstrated that the total regression error can be decomposed in bias and variance. In Section 2.3.3 we gave a strict definition of how the model complexity H can be measured. In Section 2.3.4 we defined the overfitted models as the fully trained models with the complexity H higher than the optimal value H_{opt}. The methods that prevent overfitting are called *regularization* methods. Most of them use prior knowledge about the data to decrease the model complexity H in order to decrease the variance, but some of them, like data augmentation, directly decrease the variance without affecting H.

Usually regularization methods are chosen in such a way that they do not introduce a significant bias. It is possible if we have so-called *domain knowledge*. In this case we know that the best solution $f(x)$ has a particular property, so we restrict the set of possible functions to only those functions with this property, like

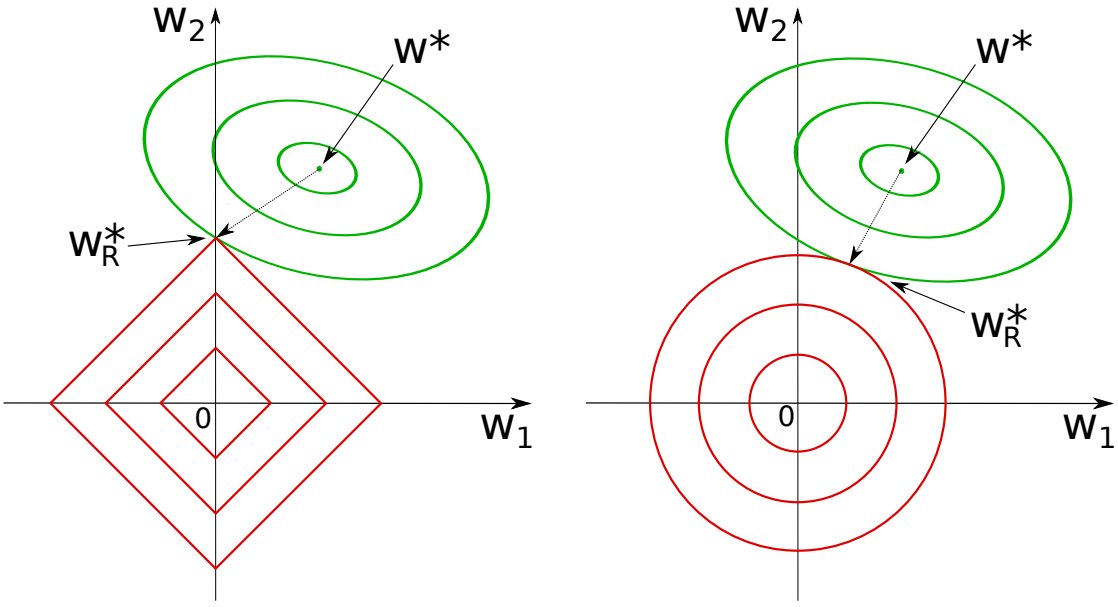
$$\begin{cases} w^* = \arg\min_w L_{min}(f(x;w)) \\ R(f) \leq C \end{cases} \tag{2.22}$$

This way we incorporate our knowledge in the learning process. It allows us to decrease the variance while keeping the bias approximately the same. Thus, the regularization methods prevent overfitting of too complex models.

Let us consider an example. In many cases of regression we know that $f(x)$ should be smooth. Formally it means that its "smoothness" should be less than a particular value, for example

$$R(f) = \int \left| \frac{\partial^m}{\partial x^m} f(x; w) \right|^2 dx \leq C$$

For $m = 1$ it restricts $f(x)$ from having too large first derivatives, while for $m = 2$ it restricts $f(x)$ from changing the first derivative too quickly. The parameter C is a hyperparameter that regulates the strength of regularization. It should be chosen by one of the methods described earlier.



(a) L_1 weight penalty function. (b) L_2 weight penalty function.

Figure 2.13: The green ellipses represent the levels of the loss function L_{train}. The red squares/circles represent the levels of the penalty function $R(f)$. The penalty function moves the non-regularized minimum w^* to the regularized minimum w_R^*, which is closer to the origin.

The proposed way of regularization is *hard*, because it sets a strict boundary on a set of possible functions. Such problems of conditional minimization are usually solved by the method of Karush-Kuhn-Tacker multipliers. It allows the solution of an unconditional minimization problem with the additional term

$$L_{min}(f) = L_{train}(f) + \lambda R(f) \tag{2.23}$$

This is a *soft* form of regularization. In this case the boundary is not strict, but the functions with lower $R(f)$ have a priority to be chosen as the best. Here λ is also a predefined hyperparameter that regulates the regularization strength. It is known from theory ([5]) that the solution of the hard regularization problem (2.22) for some λ_{min} is also a solution of the soft one (2.23), so these tasks are in some sense equivalent. In practice it is usually required to solve a soft regularization problem.

In the method of gradient descent (Equation (2.4)) soft regularization requires the computation of the derivative $\partial R(f)/\partial w$, which is added to the derivative of the training loss

$$w^t = w^{t-1} - \alpha \left(\frac{\partial L_{train}(f(w))}{\partial w} + \lambda \frac{\partial R(f(w))}{\partial w} \right) \tag{2.24}$$

2.3.6 Bayesian interpretation

It is possible to interpret regularization from another point of view. Let us consider the joint probability distribution of data and parameters $P(x, y, w)$ all together. Assuming the independence of objects in the training set, we can find the set of optimal parameters w^* using the method of maximum likelihood, i.e., find w^*, such that it yields the maximum joint probability of D_N

$$w^* = \arg\max_w P(D_N, w) = \arg\max_w P(w)P(D_N|w) = \arg\max_w P(w) \prod_{i=1}^{N} P(x_i, y_i|w)$$

Since the logarithm is a monotonically increasing function, it is more convenient to consider the negative log-likelihood of D_N:

$$w^* = \arg\min_w \left(-log\, P(D_N, w)\right) = \arg\min_w \left(-log\, P(w) - \sum_{i=1}^{N} log\, P(x_i, y_i|w)\right)$$

Let us assume that the difference $y - f(x; w)$ has a normal distribution

$$P(x, y|w) = \frac{1}{\sqrt{2\pi}\sigma} \exp\left(-\frac{(y - f(x; w))^2}{2\sigma^2}\right)$$

Then the maximum likelihood solution can be found as

$$w^* = \arg\min_w \left(-log\, P(w) + \sum_{i=1}^{N} (y_i - f(x_i; w))^2\right)$$

i.e., it is equivalent to the solution of the L_{train} minimization problem with the MSE loss function and a regularizer which depends only on the parameters $R(f) = R(w) = -log\, P(w)/\lambda + C$. Therefore, a soft regularizer can be considered as the initialization of an exponential prior distribution over a set of parameters

$$P(w) = \frac{1}{C} e^{-\lambda R(w)},\ C = \int_w e^{-\lambda R(w)} \Rightarrow \int_w P(w) = 1$$

2.4 Model selection

Since the real value of $L_{int}(f)$ is never available, we need to use its approximation $L_{test}(f)$ (Equation (2.20)). The larger are the numbers S and J, the closer is the approximation to the real value $L_{int}(f)$. However, unless the training set size N is very large, the increase of the test set size S means the decrease of N.

2.4.1 Cross-validation

A popular approach to estimate the test error is called *cross-validation*. It can be described by the following steps:

1. Split N samples on J groups of approximately the same size
2. Choose one of the group and leave it for test
3. Train a function $f(x)$ on the other $J-1$ groups
4. Evaluate the performance of the trained $f(x)$ on a chosen group
5. Repeat 2-4 for all other groups and average the test results

Cross-validation allows us to estimate $L_{test}(f)$ with $1 \leq J \leq N$ and $N/2 \leq K \leq N-1$. While the training sets D_N^j are not independent, the method still gives robust estimations of the test error, and is a de-facto standard for searching hyperparameters. The typical choice of J is $3-10$. When $J = N$, the method is called *leave-one-out* cross-validation. The main disadvantage of the cross validation is time: it requires $J-1$ times more than the training on the full dataset D_N.

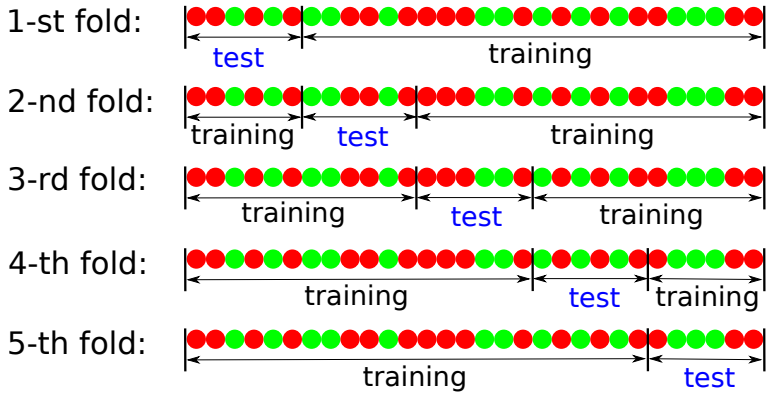


Figure 2.14: An example of 5-fold cross-validation. Training and test sets for each fold are shown.

2.4.2 AIC and BIC based approaches

While cross-validation is a simple and reliable approach, it is not the most efficient. Apart from structural risk minimization (Section 2.3.3), there exist other methods that can be used for the choice of the most appropriate model. In this

section we describe two measures that might be employed for this purpose: Akaike Information Criterion (AIC) and Bayesian Information Criterion (BIC). There also exist two other similar criteria: minimum description length (MDL) and minimum message length (MML), which differ from AIC/BIC in details.

In 1974, Hirotsugu Akaike presented a formula that is expected to obtain its minimum when applied to the best model. It contains two terms. The first one is the negative log likelihood of the observed data for the current model, which measures the quality of the model. The second one is the effective number of free parameters, which measures the price that is paid for the achieved quality. Its derivation is based on the Kullback-Leibler divergence - a (non-symmetric) measure of difference between two probability distributions:

$$D_{KL}(p||q) = \int_{-\infty}^{\infty} p(x) \log \frac{p(x)}{q(x)} dx$$

.

Let Z be the observed data, M_i the candidate model, θ_i the vector of its parameters, and $|\theta_i|$ is number of parameters. Moreover, let $\tilde{\theta}_i$ be the best vector of the parameters for this model, i.e., $\tilde{\theta}_i$ minimizes $D_{KL}(p(x)||q(x|\theta_i))$, assuming that $p(x)$ is a true p.d.f. and $q(x|\theta_i)$ is a parametrized p.d.f. in the model M_i. This is equivalent to

$$E_p \left[\frac{\partial}{\partial \theta_i} \log(q(x|\theta_i))|_{\theta_i=\tilde{\theta}_i} \right] = 0,$$

where E_p is the expectation with respect to the distribution $p(x)$.

Next, let $\hat{\theta}_i$ be the value of θ_i that minimizes the negative log-likelihood $-\log L(M_i|Z) = \log q(Z|\theta_i)$. Using Taylor expansion around the point $\tilde{\theta}_i$ and taking the expectation, we get the first order term equal to zero. After using the approximation for the second order term, we obtain the final result:

$$-\log L(M_i|Z) \approx -\log L(\hat{\theta}_i|Z) + |\theta_i|,$$

where $|\theta_i|$ is the effective dimensionality of the parameter space of the model

M_i. For our experiments we use the corrected version AIC_c

$$-\log L(M_i|Z) \approx -\log L(\hat{\theta}_i|Z) + |\theta_i| + \frac{|\theta_i|(|\theta_i|+1)}{N-|\theta_i|-1}, \tag{2.25}$$

which is necessary when the relation between dimensionality and total number of observations $|\theta_i| \ll N$ does not hold. An in-depth discussion about AIC may be found in [11].

In 1978 Schwarz presented another approach, arising from a probabilistic description of the data given a current model. This method came to be known as the Bayesian Information Criterion (BIC). It has almost the same form as the AIC, but penalizes complex models more heavily. Instead of expansion around the unknown vector $\tilde{\theta}_i$, the expansion around the known $\hat{\theta}_i$ is used. A comprehensive description of the BIC derivation may be found in [6]. Using the same notation as for AIC, we reproduce the final result:

$$\log L(M_i|Z) \approx \log L(\hat{\theta}_i|Z) + \log g_i(\hat{\theta}_i) + \frac{|\theta_i|}{2}\log 2\pi - \frac{\log |H_{\theta_i}|}{2}$$

Here $g(\theta_i)$ is the prior distribution of the parameters θ_i, and $|H_{\theta_i}|$ is the determinant of the Hessian matrix, consisting of

$$H_{\theta_i}^{mn} = -\frac{\partial^2[\log L(\hat{\theta}_i|Z) + \log g_i(\hat{\theta}_i)]}{\partial\theta_i^m \partial\theta_i^n}|_{\theta_i=\hat{\theta}_i}$$

When $g(\theta_i)$ is flat, it can be omitted here, as well as in the main formula for $\log L(M_i|Z)$, since it has zero derivative and does not influence the best model $\arg\min_i L(M_i|Z)$.

Since for independent observations

$$\log L(\hat{\theta}_i|Z) = \sum_{j=1}^{N} \log L(\hat{\theta}_i|z_j) \xrightarrow{N\to\infty} NE_p[\log L(\hat{\theta}_i|z)],$$

the expression for $H_{\theta_i}^{mn}$ can be transformed into

$$H_{\theta_i}^{mn} = -N\frac{\partial^2 E_p[\log L(\hat{\theta}_i|z)]}{\partial\theta_i^m\partial\theta_i^n}|_{\theta_i=\hat{\theta}_i} = NI_{\theta_i}^{mn},$$

where I_{θ_i} is the Fisher information matrix with non-zero determinant. Thus,

$$\log|H_{\theta_i}| = \log|NI_{\theta_i}| = \log N^{|\theta_i|}I_{\theta_i} = |\theta_i|\log N + \log I_{\theta_i}$$

The last term is $o(N)$, and for large N can be omitted. Therefore, the final formula for BIC is the following:

$$-\log L(M_i|Z) \approx -\log L(\hat{\theta}_i|Z) + \frac{|\theta_i|}{2}\log\frac{N}{2\pi} \tag{2.26}$$

2.5 Image classification

The problem of image classification is one of the earliest applications for neural networks. For example, the first successful implementation of the perceptron Mark-1 mentioned in Section 2.2.3 was constructed to classify images. In fact, it was called "perceptron" because it was able to perceive the visual information. Image classification is relatively easy for people, but quite difficult for machines. The availability of benchmark datasets made it popular for comparison of different algorithms. The image classification task was one of the drivers in the area of supervised machine learning.

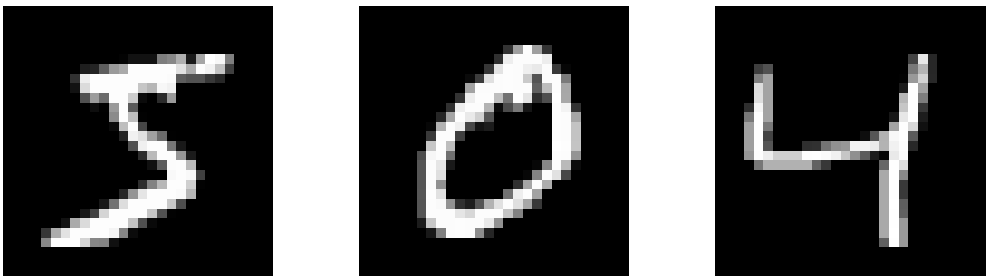
In this task the training set consists of images of the same size. Images are given as matrices of pixel intensities. Gray-scale images need only one matrix, while colored images require an independent matrix for each color channel. Image labels correspond to the object presented on an image. Usually the datasets contain images with only one object, so the images can be labeled unambiguously. To some extent, the image labels are invariant to image rotations, translations and scaling. This is one of the ways to augment the dataset in order to avoid overfitting.

One of the most popular examples is the problem of handwritten digit clas-

sification. This is a convenient task for several reasons. First, this is a well posed problem with a clear set of classes corresponding to digits from 0 to 9. Second, the task is effectively unambiguous, so the images contain enough information to achieve a near-zero classification error. Third, this is a task with the a practical outcome - after achieving sufficiently good accuracy the systems of handwritten digit classification were employed in post offices, banks, and other organizations.

2.5.1 Benchmark datasets

The MNIST benchmark dataset [61] with $60k$ gray-scale images of handwritten digits of size 28×28 was released 1998. Since then it remained the main tool for evaluation of new algorithms for image classification. The test error of modern algorithms on MNIST is less than 0.3%. This is comparable with a human error rate, so there is practically no space for further improvement. For this reason it is not considered as a reliable benchmark anymore, but is still used as a toy example.



(a) (b) (c)

Figure 2.15: Some images from the MNIST dataset

Another popular benchmark dataset for image classification is CIFAR [53], presented in 2009. It contains $50k$ colored images of size 32×32 pixels. The objects on the images are different vehicles, animals, and others. There are two versions of its labeling: on 10 broad classes, and on 100 more specific classes. This dataset has a similar size to MNIST, but is more difficult for classification.

Currently the best algorithms demonstrate around 8% test error for 10 classes, and 36% for 100 classes. Since the test error is not vanishing, the dataset is more reliable for algorithm comparison.

(a) (b) (c)

Figure 2.16: Some images from the CIFAR dataset

The SVHN (Street View House Numbers) dataset [76] was presented in 2011. It contains over 600k colored images with digits from house numbers recorded during the Google Street View project. The size of the images is the same as in the CIFAR dataset: 32×32. Unlike MNIST, the SVHN dataset is composed of images with a much larger variety of digit shapes, sizes and locations. Some of the images also contain parts of other digits. For this reason the dataset is actively used as a benchmark. The lowest achieved test error rate is around 2%.

(a) (b) (c)

Figure 2.17: Some images from the SVHN dataset

ImageNet [16] is the largest available public dataset of images, collected from the Internet, and annotated using Amazon Mechanical Turk. It contains over 15 million colored high resolution images with more than 22 thousand labels. The annual LSVRC (Large Scale Visual Recognition Challenge) competition is based on the ImageNet dataset. The rise of interest in neural networks happened after LSVRC-2012, when the deep neural network won with the test error around 15% [54], while the second best solution demonstrated an error of around 26%.

2.6 Conclusion

In this chapter we presented the background information required for further discussion.

In Section 2.2 we presented an overview of neural networks. We started from the model of a single neuron, and ended by a description of recurrent neural networks. We provided the definitions for multiclass perceptrons and multilayer perceptrons, discussed nonlinear activation functions, introduced a convenient matrix notation, and described the backpropagation algorithm, which is used for training. In the end we presented an overview of recurrent neural networks, which are used for learning from sequences. These concepts and definitions serve as a foundation for the discussions in the following chapters.

In Section 2.3 we discussed the problem of overfitting. It is a common problem of most supervised learning models, but it is especially important for neural networks due to the large number of learned parameters. We derived the bias-variance decomposition for the regression problem, described the problem of bias-variance tradeoff, provided the definition of model complexity, and presented the concepts of underfitting and overfitting. Using the introduced notation, we defined regularization methods as a set of restrictions, that decrease model variance without a significant decrease of model bias, that allows us to decrease the expected test error. We also noted that most of regularization methods are based on the domain knowledge, that is thus incorporated in the learning process. The Chapters 3 and 4 describe regularization methods from

this point of view.

The last two Sections 2.4 and 2.5 present the background information for the experimental parts of Chapters 5 and 4 accordingly.

Chapter 3

Regularization of neural networks

3.1 Introduction

As it was shown in Section 2.2.10, neural networks can approximate any continuous function with a required precision. This is the reason why they are used in the most difficult classification tasks with a large number of features, that appear in the areas of image, video, and audio processing. In these areas the data have an important property: the features are ordered in one or more dimensions, and the recognized objects are continuous in these dimensions. For example, image objects are continuous in width an height, while speech is continuous in time. This is a very important property that has to be used in regularization. Moreover, usually the classification objects are invariant to some extent to variations in these dimensions, such as translation and scaling. This information should also be taken into account. We describe a number of regularization methods and explain how they deal with overfitting.

A good overview of regularization methods for neural networks is given in [8]. It describes the techniques developed up to 2006, when the book was released. However, a number of new efficient methods appeared after that. Many of them are described in a yet unpublished book "Deep Learning" [3]. All of these methods are discussed in this chapter.

3.2 Standard regularization

One of the simplest and most popular choices of the regularization term is the L^p norm of the set of parameters

$$R(w) = \frac{1}{p}||w||_p^p,\ p = \{1, 2\} \tag{3.1}$$

This is a general method of weight penalization, that is used not only with neural networks, but also with a variety of other models. As it was shown in Section 2.3.5, it initializes a prior distribution over the set of parameters w, encouraging the weights to be as small as possible until supported by data. It effectively forbids parameter values that are too far from the origin, and therefore reduces the variance.

If $p = 2$, the method is known as *weight decay*. This is a version of Tikhonov regularization [99] with the Tikhonov matrix $\Gamma = I$. The method is simple to implement: the corresponding derivative for the gradient descent update rule (Equation (2.24)) is

$$\frac{\partial R(w)}{\partial w} = \frac{1}{2}\frac{\partial ||w||_2^2}{\partial w} = w$$

For $p = 1$ the method is known as *LASSO* (least absolute shrinkage and selection operator) [98]. Later we show, that if the corresponding coefficient λ is large enough, some of the elements of w become zero, making the vector w *sparse*. This is a convenient property that allows the removal of some perceptron connections and simplifies calculations. The derivative of $R(w)$ is

$$\frac{\partial R(w)}{\partial w} = \frac{\partial ||w||_1}{\partial w} = sign(w)$$

Both options add a term that is convex, so if the main component of L_{min} is convex as well, they allow the application of methods of convex optimization.

Let us consider w^* as the solution of the unregularized problem of $L_{train}(w)$ minimization. $L_{train}(w)$ has its Hessian matrix $H_{ij} = \partial^2 L_{train}/(\partial w_i \partial w_j)$, evaluated at w^*. By the definition of H and w^*, we can conclude that

$$\nabla_w L_{train}(w) = H(w - w^*) \Rightarrow \nabla_w L_{train}(w^*) = 0 \tag{3.2}$$

Now let $\hat{w}$ be the solution of the regularized problem (3.1) with $b = 2$. From (2.24) we thus obtain, that for $\hat{w}$

$$H(\hat{w} - w^*) + \lambda\hat{w} = 0 \Rightarrow \hat{w} = (H + \lambda I)^{-1}Hw^*, \tag{3.3}$$

i.e., the L^2 regularization transforms the solution w^* by multiplication on $(H + \lambda I)^{-1}H$. The Hessian H has a positive determinant, so it can be represented using eigenvalue decomposition as $H = Q\Lambda Q^{-1}$, where Λ is the diagonal matrix with eigenvalues, and Q is the orthonormal matrix with corresponding eigenvectors. Note, that $Q^T = Q^{-1}$. Using this decomposition in (3.3), we get

$$(Q^T\hat{w}) = (\Lambda + \lambda I)^{-1}\Lambda(Q^T w^*) \tag{3.4}$$

Thus, in the orthonormal basis given by Q^T, L^2 regularization looks like rescaling the eigenvalues of Λ, because the eigenvalues of $(\Lambda + \lambda I)^{-1}\Lambda$ are $\frac{\Lambda_i}{\Lambda_i+\lambda}$. It causes the rescaling of the components of $Q^T w^*$, such that the elements corresponding to large eigenvalues $\Lambda_i \gg \lambda$ remain almost the same, while the components corresponding to $\Lambda_i \ll \lambda$ are pushed towards zero. The eigenvalues Λ_i represent the curvature of L_{min} in the direction of their eigenvectors, so regularization decreases the components of $Q^T w^*$ that do not significantly affect the values of L_{min}.

We can perform similar derivations for L^1 norm, that gives an equation

$$H(\hat{w} - w^*) + \lambda\, sign(\hat{w}) = 0 \tag{3.5}$$

For simplicity we assume that the matrix H is already diagonal with the elements Λ_i. In this case Equation (3.5) is equivalent to the system of equations for each $\hat{w}_i$ component:

$$\Lambda_i(\hat{w}_i - w_i^*) + \lambda\, sign(\hat{w}_i) = 0$$

Given that $\hat{w}$ and w^* are close, we can assume that $sign(\hat{w}) = sign(w^*)$. It gives us the equation

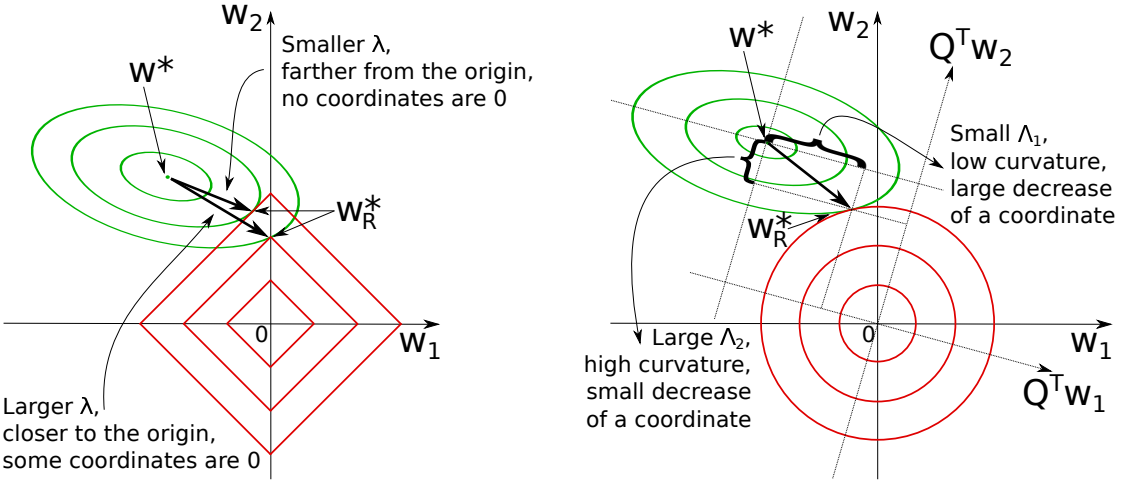
$$|\hat{w}_i| = |w_i^*| - \frac{\lambda}{\Lambda_i}$$

However, if $|w_i^*| < \frac{\lambda}{\Lambda_i}$, the solution does not exist. In this case the closest approximation is $\hat{w}_i = 0$, so the final solution is given by

$$\hat{w}_i = sign(w_i^*) \max(|w_i^*| - \frac{\lambda}{\Lambda_i}, 0)$$

Thus, regularization shifts the coordinates of w^* close to 0 on λ/Λ_i if they are larger than this value, and makes them zero otherwise. While the L^2 norm pushes the components with low curvature towards 0, L^1 essentially assigns them to 0, making the representation of $\hat{w}$ sparse.

As we have shown in Section 2.3.6, soft regularization corresponds to the exponential prior over the set of parameters. If $p = 2$, then we get a Normal distribution with zero mean and $1/\sqrt{\lambda}$ standard deviation. If $p = 1$ then the distribution is Laplacian with $b = 1/\lambda$.



(a) L_1 weight penalty function.

(b) L_2 weight penalty function.

Figure 3.1: Effects of L_1 and L_2-norm regularization. When L_1 is used, large λ makes some coordinates equal to 0. L_2-norm scales the coordinates depending on the corresponding eigenvalues.

3.2.1 Drawbacks

In the majority of tasks the output of the neural network should be invariant to the scaling of the input vector. For the NN without regularization it leads to the corresponding scaling of the first layer weights, so the network output remains the same. However, if a prior distribution is used for all weights together, it is not true anymore. This happens because all weights give the same contribution to the regularization term. In order to deal with it, independent regularizers for each layer might be used, i.e.,

$$\lambda_i R_i(w) = \frac{\lambda_i}{b}||w_i||_b^b, i = 1, \ldots, N,$$

where N is the number of layers. Independent regularizers for each layer solve the linear invariance problem, but introduce N new hyperparameters that need to be tuned. This is a significant drawback of standard regularization applied to neural networks.

Another drawback of soft regularizers is the chance of getting stuck in local minima induced by a low value of the additional term. In this case it is recommended to use hard constraints. Hard constraints are also preferable when a large learning rate is used. If a large rate causes divergence from the minimum, they make it possible to stop the process once the weight values reach the limit.

3.3 Early stopping

Early stopping assumes that the iterative procedure of L_{min} minimization, such as stochastic gradient descent, is stopped before the function L_{min} reaches its minimum. If the complexity of the model $M > M_{opt}$, it might prevent overfitting.

Typically a neural network is initialized with small weights with the maximum value 0.01 - 0.1, sampled from the distribution with zero mean. This has two consequences.

1. Small values of weights mean that the training procedure is mostly increasing their absolute values. Early stopping prevents this increase, and

therefore serves as a method of weight decay, that has been considered in Section 3.2.

2. For reasons of symmetry, initial weights with zero mean cause the input values of neurons to be close to zero for most of the input vectors. Thus, the network acts as a linear nilpotent operator with a very low complexity. During the process of training, the network becomes more and more non-linear, gradually increasing its complexity. Hence, early stopping prevents the network from being too complex.

We can think about the number of training iterations as a special hyperparameter that is changing during the training procedure, so early stopping is the process of tuning this hyperparameter.

In practice the moment of stopping is defined by the dynamics of error on the *validation set*, which is a part of a training set not used for training. For this reason it gives an empirical estimation of L_{test}. Once the error on the validation set starts to improve, the algorithm stops and returns the set of weights corresponding to the lowest validation error. In practice, validation error is not computed every iteration, and it is required that $p > 1$ consecutive evaluations be worse than the best value to stop the algorithm. Moreover, it is also possible to use the actual loss function of interest on the validation set (like classification error), instead of a differentiable, but artificial loss function (like MSE, or log-likelihood). In many cases these functions achieve their minimum at different points.

Once the optimal number of iterations has been defined, the validation set might be used for training to further improve the accuracy. There are two strategies of how it can be done. The first one assumes retraining of the whole dataset with a defined number of iterations. However, since this time the dataset is larger, it might not be enough to reach the optimum. Moreover, it requires spending the same amount of time on retraining. Another more popular approach, assumes the continuation of training on the full dataset until the *training error* on the validation set reaches the level of the *training error* achieved on the training set when the process was stopped.

The effect of restricting the absolute values of weights can be demonstrated

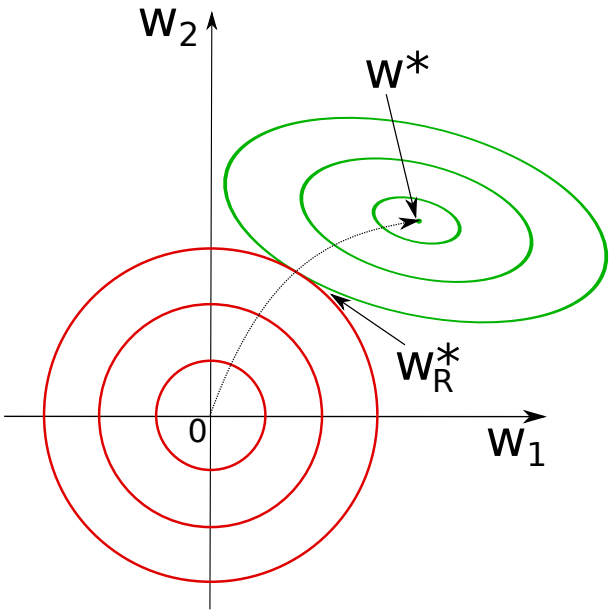


Figure 3.2: Early stopping prevents the solution from being too far from the origin, thus working similar to L_2 regularization, which scales the coordinates.

mathematically. Using the update rule (2.4) and the same Hessian matrix H as before, we can get

$$w^t - w^* = w^{t-1} - \alpha H(w^{t-1} - w^*) - w^* = (I - \alpha H)(w^{t-1} - w^*) \tag{3.6}$$

This is a recursive formula for the sequence $w^t - w^*$. Applying it t times and assuming that $w^0 = 0$, we get

$$w^t = (I - (I - \alpha H)^t)w^*$$

In the orthonormal basis Q^T, where $H = QTQ^{-1}$ is the eigendecomposition of H, T is a diagonal matrix with the eigenvalues Λ_i. The training procedure converges, if $\forall i \, |1 - \alpha\Lambda_i| < 1$. If the eigenvalue is very small, i.e., $\Lambda_i \ll 1$, then we can drop all terms of the Taylor approximation except the first two

$$w_i^t \approx (1 - (1 - \alpha t\Lambda_i))w_i^* = \alpha t\Lambda_i w_i^*$$

Comparing this with the regularization effect of L^2, i.e., $\hat{w}_i = \Lambda_i/(\Lambda_i + \lambda)$, we can conclude that under the stated assumptions early stopping is equivalent to L^2 regularization with the parameter

$$\lambda = \frac{1}{\alpha t} - \Lambda_i \approx \frac{1}{\alpha t}$$

As expected, the larger t is, the smaller is the regularization effect. Recalling that the eigenvalues of H correspond to the curvature of L_{min} in the directions of the eigenvectors, we can conclude that early stopping prevents learning from descent in the directions of low curvature.

3.4 Convolutional layers

As we discussed in Section 3.1, in some tasks like image classification the network predictions should be invariant to the translation of the input vector. Therefore, it is reasonable to expect that the object representation in the hidden layer does not depend on its location. While it is not easy to achieve absolute invariance, it is easy to make the hidden layer representation move together with the object itself. Two things are required to achieve this property:

- The weights of connections shifted on the same distance in the input and hidden vectors should be equal.
- The hidden layer value should depend on the limited amount of input vector elements much smaller than the vector size.

We provide a brief explanation for a one-dimensional case. Let us consider the input vector V of length N, and the subset of $M < N$ consecutive elements in the middle of V, that we call an object of interest D. Next, let each element of the hidden layer H depend only on $K << M$ consecutive elements of V. Then its size is $N - K + 1$. We call a subset S of $M - K + 1$ consecutive elements of H a *representation* of D, if each of its elements depends only on the elements of D.

Let us suppose, that the object D was shifted by some distance d, such that all its elements remained inside the vector V. Then there exists another subset

$\tilde{S}$, shifted by d with respect to S, that is a representation of a shifted object D. It has the same length $M - K + 1$ as S. Since the weights of connections, shifted on the same distance, are equal, the subset $\tilde{S}$ has the same values as S. Thus, the shift of the object D leads to the shift of its representation S by the same distance.

Thus, the procedure of hidden layer computation looks like an iterative application of a filter of size K to every possible position in the vector V. This is a well known operation in mathematics called *filtering*, or *convolution* (if applied to an inverse filter). A hidden layer computed this way is called *convolutional*.

While the initial goal is to keep the invariant representation, these restrictions also lead to a dramatic decrease of the model complexity. If the input vector has size N, and the hidden layer has size $N - K + 1$, then the number of weights (with biases) is $N \times (N - K + 2)$. Restrictions allow a decrease in the number of weights to $K + 1$, i.e., much less than the initial number. Thus, they significantly reduce the model variance.

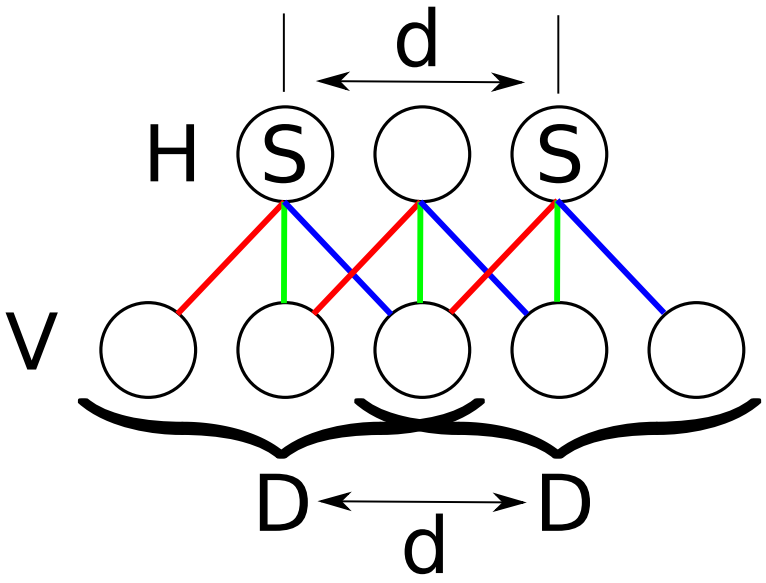


Figure 3.3: 1-dimensional convolution layer. Same colors represent the same weights. The shift of the object D on d elements leads to the shift of the representation S on d elements as well. The number of weights is reduced from $5 \times 3 = 15$ to 3.

3.4.1 Implementation details

Let us assume that a function $f_i(x; w)$ (Equation (2.9)) implements a convolutional layer. Since convolution is a specific type of linear transformation, it can also be implemented as a multiplication on a specific matrix of weights. However, it is not efficient, because it requires the performance of a lot of unnecessary multiplications on zeros and the storage of them in memory.

Let us use the same notation as earlier: the bottom index corresponds to the layer number, and the top index indicates an element in a vector. Then the convolution layer can be computed as

$$y_i^j = \sum_{k=1}^{K} y_{i-1}^{j+k-1} w_i^k + b_i,\ j = 1, \ldots, N, \tag{3.7}$$

where the size of the vector y_{i-1} is $N + K - 1$. This way of computation involves only necessary operations.

On the backward pass we can avoid matrix multiplication as well. Let us assume that we know the gradients $dy_i^j = \partial L / \partial y_{i-1}^j$ for the layer activations y_i^j. Then using the chain rule, we can get

$$\begin{aligned} dy_{i-1}^j = \frac{\partial L}{\partial y_{i-1}^j} &= \sum_{k=max(1,K+1-j)}^{min(K,K+N-j)} \frac{\partial L}{\partial y_i^{j+k-K}} \cdot \frac{\partial y_i^{j+k-K}}{\partial y_{i-1}^j} = \\ &= \sum_{k=max(1,K+1-j)}^{min(K,K+N-j)} dy_i^{j+k-K} \cdot w_i^{K-k+1},\ j = 1, \ldots, N+K-1 \end{aligned} \tag{3.8}$$

We can see the similarity between Equations (3.7) and (3.8). In both cases each element of the output is the sum of products of K consecutive elements of the input multiplied by the elements of weights w. Thus, the implementation of the reverse function $\tilde{f}_i$ is also filtering of its input dy_i with the vector of weights w_i, with some modifications that we discuss later.

Applying the chain rule (Equation (2.8)) to Equation (3.7), we obtain the

derivatives for weights

$$\begin{aligned} dw_i^k &= \frac{\partial L}{\partial w_i^k} = \sum_{j=1}^{N} \frac{\partial y_i^j}{\partial w_i^k} \cdot \frac{\partial L}{\partial y_i^j} = \sum_{j=1}^{N} y_{i-1}^{j+k-1} \cdot dy_i^j,\ k = 1, \ldots, K \\ db_i &= \frac{\partial L}{\partial b_i} = \sum_{j=1}^{N} \frac{\partial y_i^j}{\partial b_i} \cdot \frac{\partial L}{\partial y_i^j} = \sum_{j=1}^{N} dy_i^j \end{aligned} \tag{3.9}$$

We see that each element of dy_i (Equation (3.7)) is a sum of products of consecutive elements of y_{i-1} with all elements of dy_i. Thus, weight gradient computation also corresponds to filtering of the vector y_{i-1} by the vector dy_i.

3.4.2 Forward - backward duality

While reverse functions $\tilde{f}_i$ also perform filtering, they have some difference. The first one is the order of weights iteration. If on the forward pass the order is direct, on the backward pass it is the reverse. In other words, if on the forward pass we perform filtering, on the backward pass we perform convolution.

The second difference is border conditions. Equation (3.7) represents *valid* filtering. It means that each element of output is given by a sum of products of **all** K weights with some K elements of the input. If the length of the input vector is N and the filter is K, then the length of the output is $N - K + 1$, i.e., the number of different sets of consecutive elements of size K in the vector of size N. In contrast, Equation (3.8) represents *full* filtering. In this case the output is given as a sum of products of **at least one** of K weights with at least one element of input. Using the same notation as before, in this case the output length is $N + K - 1$. Moreover, the situation might be the opposite. If on the forward pass the operation is full, on the backward pass it is valid.

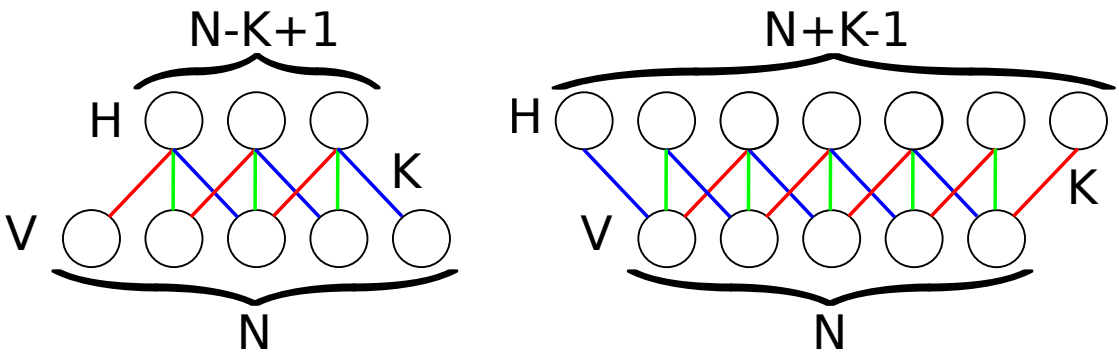
This property might be generalized. Notice, that the full operation is equivalent to the valid operation, if the input vector is joined with $K - 1$ zeros from both sides. These sets of zeros are called *padding*. Thus, zero size padding on the forward pass leads to the $K - 1$ padding on the backward pass. Since the dependency between padding sizes is linear, we get that for an arbitrary padding

$0 \leq p \leq K-1$ its corresponding padding is given by the formula

$$\tilde{p} = K - 1 - p \tag{3.10}$$

The calculation of the weights in (3.9) is performed with the same padding p.

It is easy to notice that the demonstrated differences are *dual*. Indeed, double reverse order is equivalent to the original order. According to Equation (3.10), double reverse padding is equivalent to the original padding. Thus, the functions f_i and $\tilde{f}_i$ performing the convolutional transformations on the forward and backward passes are also dual. In fact, the duality of f_i and $\tilde{f}_i$ can be demonstrated in another way. For the linear function f_i with weights, its corresponding function $\tilde{f}_i$ is the multiplication of the argument on the transformed matrix of weights. Since the operation of transposition is dual, the functions f_i and $\tilde{f}_i$ are also dual.



(a) Valid filtering. (b) Full filtering.

Figure 3.4: The difference between valid and full filtering for $N = 5$ and $K = 3$.

3.4.3 Two-dimensional convolution

Let us consider the image classification problem. In this case the input vector consists of one (or several) two-dimensional array(s). Thus, we can also consider the hidden layer as a set of two-dimensional arrays. In the neural network literature these arrays are known as *feature maps*.

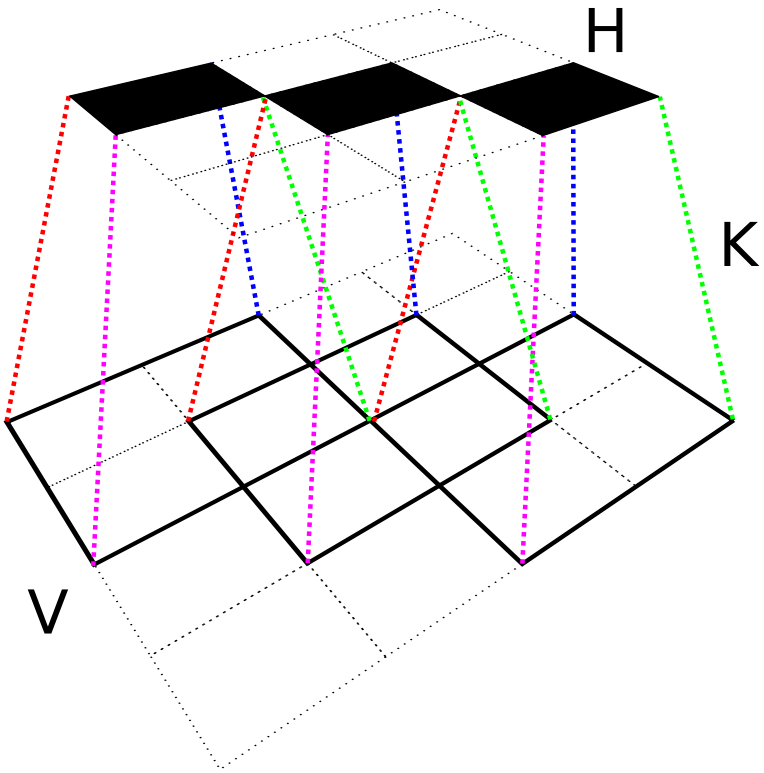


Figure 3.5: 2-dimensional convolution layer. The 2×2 kernel is applied to the 4×4 input feature map in order to get a 3×3 output map. The number of weights is reduced from $4 \times 4 \times 3 \times 3 = 144$ to 4.

In the image classification task we are interested in invariance to translation in both directions: horizontal and vertical. In order to achieve this, we need to consider two-dimensional convolution. Let an image be the size $N \times N$. Similar to Equation (3.7), a map of convolutional layer is computed as

$$y_i^{j_1,j_2} = \sum_{k_1=1}^{K_1} \sum_{k_2=1}^{K_2} y_{i-1}^{j_1+k_1-1,j_2+k_2-1} w_i^{k_1,k_2} + b_i,\ j_1, j_2 = 1, \ldots, N \tag{3.11}$$

Thus, two-dimensional convolution involves all possible square subregions of an image of size $K_1 \times K_2$. Usually in practice the filters are chosen to be squared, i.e., $K_1 = K_2$. Their values usually vary from 2 to 7.

Equation (3.11) represents convolution of a single feature map. In general

the number of feature maps on the layers $i-1$ and i can be arbitrary. In this case each pair of feature maps requires its own filter. Hence, if the numbers of feature maps are S_{i-1} and S_i, then the total number of filters is $S_{i-1} \times S_i$, and the total number of weights is $K_1 \times K_2 \times S_{i-1} \times S_i + S_i$. Additional S_i weights correspond to biases added to each output feature map. The number of feature maps used in practice might be up to several hundred.

In Section 3.4.2 we described the duality properties of convolutional layers in a one-dimensional case. Taking dimensionality into account, they remain the same in the two-dimensional case as well.

3.5 Subsampling layers

Convolutional layers allow the object representation on the higher layer to be preserved after translation, but the representation itself is also translated. Thus, it does not solve the problem of invariant representation. *Subsampling* (or pooling) layers can partially help to deal with it.

Let us again consider a one-dimensional case of a layer y_{i-1} with N neurons in it. Subsampling layers have two parameters: a sample size K and a stride S. They split the vector y_{i-1} on the blocks of a maximum K consecutive elements, on the distance S between each other. The maximum over each block gives a single value for the new layer y_i. Formally the transformation function f_i is computed as

$$y_i^j = \max_{k=1}^{min(K,N-Sj+1)} y_{i-1}^{k+Sj-1},\ j = 1, \ldots, \lceil N/S \rceil$$

Thus, the size of the new layer is $\lceil N/S \rceil$. In general, the pooling function might be different, but maximum is the most common choice.

When the derivatives dy_i are computed, they are propagated back to those elements of dy_{i-1} that give the maximum value in a block. Other gradients are assigned to zero. If several elements have the maximum value, all of them get the gradient from dy_i. If some element has the maximum value in several blocks (when $S < K$), its gradients are summarized. In the two-dimensional case subsampling is performed over the squares of the size $K \times K$, and the rules of gradient propagation remain the same.

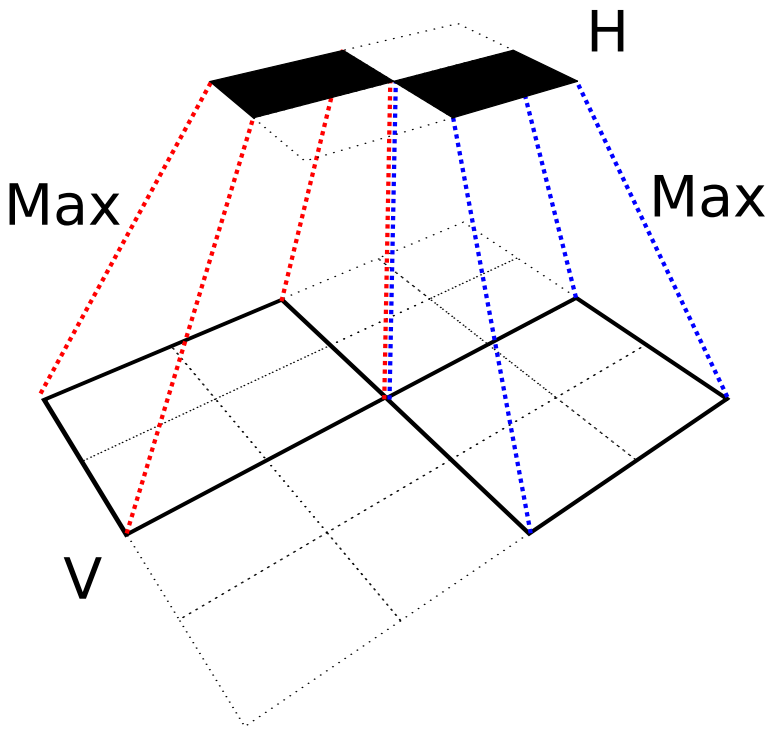


Figure 3.6: 2-dimensional subsampling layer. It chooses the maximum over the $K = 3 \times 3$ blocks, with the stride $S = (1, 1)$. Unlike the convolutional layer, it does not contain trained parameters at all.

If an object is shifted by a small distance $d < K/2$, the maximum values over a block are likely to be the same for all blocks. However, since some maximum values might be located near the border of a block, it is not guaranteed. Subsampling also provides a partial invariance of the representation to scaling, but only in the regions near the scaling center. Note that subsampling layers do not have weights, and therefore they do not increase the model complexity. Additionally they decrease the data dimensionality by S times ($S \times S$ times in the two-dimensional case), which allows the use of fewer weights on the upper level of a neural network.

Subsampling layers are widely used in image classification. The typical choice of values K and S varies from 2 to 4. Typically networks contain several (sometimes up to 10) convolutional and subsampling layers following each other. This way they achieve greater robustness to translations of the input

vector. Networks with this architecture are called *convolutional neural networks* (CNN).

3.6 Data augmentation

Another popular method of dealing with overfitting in classification problems is data augmentation. It assumes the generation of new objects of the training set using some invariant transformation function $g(x;\theta)$, applied to the existing objects in the training set. The function should preserve the object label for some range of parameters $\theta \in \Theta$. As an example we can come back to the problem of image classification. The objects presented on an image are invariant to a number of transformations, such as translation, rotation, scaling, elastic deformation, contrast, etc. All of them can be modeled by some function $g(x;\theta)$, that can generate new images with the same object, and thus the same label.

The range of possible parameters Θ is different for each transformation function. For example, the rotation of an image with the number 6 on 180 degrees makes it look like 9, therefore changing its label. The functions $g(x;\theta)$ are also specific for each domain. However, transformations such as addition of a Gaussian noise can be applied to every task, where the labels are expected to be continuous with respect to the input, i.e., to almost everyone. We will consider this later in Section 3.7. In some cases we know the type of invariant transformation, but its corresponding function is unknown or too complicated. For example, it is true for out-of-plane rotations of objects on images.

The generation of new objects with functions $g(x;\theta)$ is a simple way to incorporate knowledge about the problem into the learning algorithm, which is a goal of most regularization methods. It also has a clear explanation for the point of view of bias-variance decomposition: the larger the dataset size, the smaller the variance term, while the bias remains the same. Note that unlike other regularization methods, it does not reduce the model complexity.

The main drawback of data augmentation is increased learning time. Depending on the number of generated samples it might be many times longer than for the original dataset. For example, in the ImageNet 2012 winning sub-

mission [54] the authors generated 10 additional images for each original one: 5 of them corresponded to the center and image corners, and the other 5 were their horizontally flipped versions. Thus, the total time increased 11 times. However, for modern GPU implementations the increased learning time is not a big problem.

In fact, there is another approach that is more popular in training neural networks. Since backpropagation does not need the all data at the same time like SVM, it is possible to generate additional training objects on the fly on each iteration of backpropagation. This allows us to deal with multiple invariant functions $g(x;\theta)$ at the same time. For each of them we generate a random value of θ, which is used to obtain a new object. This allows us to avoid storing all the additional data in memory, which might be simply impossible.

Data augmentation might be non-trivial. For example, in [54] the authors generated additional images changing the light conditions in the following way. They considered all pixels as RGB-vectors in all images in the training set, and performed principal component analysis (PCA), which gives 3 eigenvalues $(\lambda_1, \lambda_2, \lambda_3)$, corresponding to eigenvectors (p_1, p_2, p_3). Then each image could be transformed by adding the a vector

$$(p_1, p_2, p_3)(\alpha_1\lambda_1, \alpha_2\lambda_2, \alpha_3\lambda_3)^T,\ \alpha_i \sim N(0, \sigma^2),$$

where the typical choice is $\sigma = 0.1$. Thus, this scheme adds a small RGB-vector to each pixel such that the colour distribution in a new image is likely to be in the training set.

As we described in Section 3.8, dropout can also be considered as data augmentation with samples, corrupted with noise. In this case the function $g(x;\theta)$ is not domain specific.

The function $g(x;\theta)$ with the distribution of $\theta \in \Theta$ induces a probability distribution over the subspace of (X, Y) and works as a generative model. One might have a temptation to build a generative model over the whole space of (X, Y), to be able to augment the dataset with more samples. However, this is meaningless, because in this case the trained classifier will just approach the Bayesian classifier that directly follows from the generative model.

3.7 Noise injection

Similar to data augmentation, one can make the classifier more robust to the variations in input vector by adding some random noise. It can be implemented by adding noise on the fly before each iteration of the backpropagation algorithm. The type of noise might be different, but the most common choice is the Gaussian distribution. In this case it is possible to show ([7]) that it is equivalent to Tikhonov regularization [99].

As before, let $l(y, f(x))$ be the mean squared error (2.15) for a single pair (x, y). The added noise vector μ with the distribution $P = N(0, \sigma^2 I)$ modifies the target function the following way:

$$l_{min} = E_P \left[(y - f(x + \mu))^2 \right]$$

For this type of noise $E_P[\mu] = 0$ and $E_P[\mu^T \mu] = \sigma^2 I$. Using the approximation $f(x + \mu) \approx f(x) + \nabla_x f(x) \cdot \mu^T$, we can derive that

$$\begin{aligned} l_{min} &\approx E_P \left[(y - f(x) - \nabla_x f(x) \cdot \mu^T)^2 \right] = \\ &= E_P \left[(y - f(x))^2 + (\nabla_x f(x) \cdot \mu^T)^2 - 2(y - f(x)) \nabla_x f(x) \cdot \mu^T \right] = \\ &= (y - f(x))^2 + E_P \left[\mu \nabla_x f(x)^T \nabla_x f(x) \mu^T \right] - 2(y - f(x)) \nabla_x f(x) \cdot E_P \left[\mu^T \right] = \\ &= (y - f(x))^2 + \sigma^2 Tr(\nabla_x f(x)^T \nabla_x f(x)) = l(y, f(x)) + \sigma^2 ||\nabla_x f(x)||_2^2 \end{aligned} \tag{3.12}$$

For the whole training set the additional term is equal to the average over all the training objects

$$R(f(x; D)) = \frac{\sigma^2}{N} \sum_{i=1}^{N} ||\nabla_x f(x_i)||, \; (x_i, y_i) \in D$$

Thus, Gaussian noise effectively requires the algorithm to have small derivatives of the output with respect to the input vector x. It means that the output becomes more robust to variations in the input vector - a property that is often desirable in real applications. It was shown that neural networks might be quite sensitive to noise [96, 33]. In Chapter 4 we will show how to efficiently

solve this problem without approximations, choosing a specific noise for each training object.

After being processed by hidden layers, random noise is not random anymore. It thus make sense to also add Gaussian noise directly to the hidden layers, inducing the robustness of each of them. It has been shown to be quite efficient in unsupervised and supervised deep learning models [83].

3.7.1 Injection to weights

It is also possible to inject Gaussian noise to weights. As expected, such injection makes the classifier robust to variations in weights. Indeed, assuming the same noise model P as before and using the approximation $f(x, w_P) \approx f(x, w) + \nabla_w f(x, w) \cdot \mu^T$, we can perform the same derivations as Equation(3.12) and get

$$l_{min} \approx l(y, f(x)) + \sigma^2 ||\nabla_w f(x, w)||_2^2$$

Note, in the case of the linear perceptron $\nabla_w f(x, w) = 0$, so this type of regularization does not affect the classifier.

Gaussian noise for weights was shown to be useful for recurrent neural networks [45, 36], but in the case of feedforward neural networks it is not very popular. In Section 3.8 we will consider noise with a Bernoulli distribution, which has been shown to be one of the most efficient regularizers.

3.8 Dropout

One of the most popular methods of regularization is called *dropout* [38, 93]. It was invented by the group of Geoffrey Hinton relatively recently (in 2012), and immediately became a compulsory tool in training neural networks. This method is very simple to implement, and demonstrates a significant improvement on a large variety of tasks.

Let us consider a multilayer perceptron with the layers $y_0, \ldots, y_N$. Then dropout on the layer y_i of the size M can be described in four steps:

1. For each training case generate a vector of length M, where each element is sampled from the Bernoulli distribution with the probability $0 < p < 1$.
2. On the forward pass multiply the values of y_i by this vector.
3. On the backward pass multiply the gradients dy_i by this vector.
4. During the test stage multiply all values of y_i by p.

Here we assume element-wise multiplication. The actual idea of dropout is a neuron removal. In the case of a supervised neural network it means multiplication by 0, but in some other models the operation might be different.

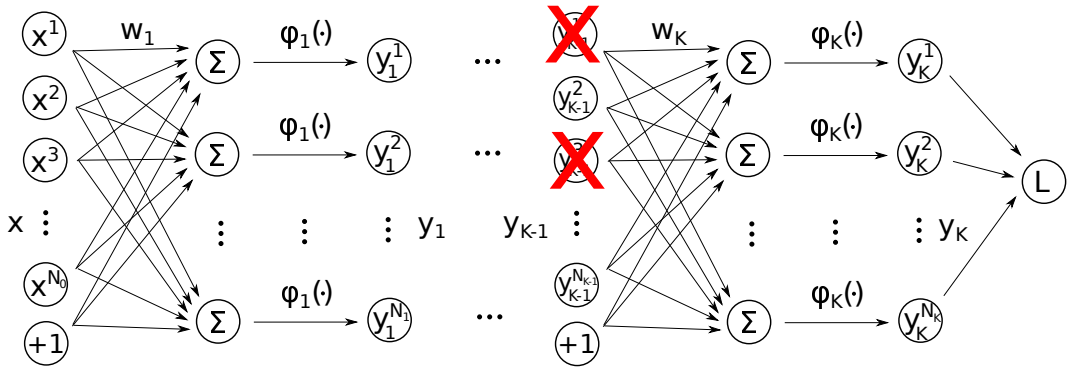


Figure 3.7: Dropout randomly 'removes' neurons from the layer with a probability $1 - p$, making neuron output values equal to 0. On the test stage the neurons are not removed, but the output weights are multiplied by p.

Dropout is typically applied to the **input** of fully connected layers, i.e., **after** the transformation of the previous layer by a non-linear activation function. Typically dropout is applied to the input of the output layer, but can also be applied to the input of all hidden layers, including the input vector. The standard choice of the dropout probability is $p = 0.5$, but for the input vector it must be less. The value $p = 0.5$ was shown to be the best in most situations, so the method does not typically require hyperparameter tuning. This is one of its advantages.

The regularization effects of dropout can be visualized by the observing particular neurons. In order to do that, one needs to solve the optimization problem

[56]

$$x^* = \arg\min f(x; w), \text{ subject to } ||x||_2 = 1,$$

where $f(x; w)$ is the output of a particular neuron, and w is the set of all weights that affect its output. In other words, it finds the input that minimizes (maximizes) the neuron input. In [93] the authors visualized the hidden layer of autoencoder trained on the MNIST dataset. Without dropout the neurons detect features close to white noise, while with dropout they detect reasonable features such as strokes, edges and spots in different image locations. Moreover it significantly decreases the mean neuron value over data, and the portion of non-zero neurons across all hidden layers. Dropout was also found to be most efficient when applied with the max-norm restriction on the weights in Equation (2.22), i.e., when $R(f) = ||w||_{\text{inf}} = \max_w w$.

3.8.1 Interpretations

Dropout can be considered as a special way of averaging different models of neural networks. Dropping a neuron is equivalent to dropping all its outgoing connections. For a large enough network the combination of dropped neurons is almost surely unique, so each input object trains its own network, which share all other non-dropped weights with other networks. As pointed out in [38], dropout can be also viewed as a way to prevent co-adaptation in neurons, so they cannot rely on the values presented by other neurons.

Dropout can also be seen as a way to add random noise to a hidden layer [93]. In some cases this noise can be marginalized, which leads us to a deterministic loss function. For example let us consider the problem of linear regression

$$w^* = \arg\min_w ||y - Xw||_2^2$$

Then dropout is an element-wise multiplication of X on the random matrix $R_{ij} \sim$ Bernoulli(p) of the same size as X. Since R is random, w^* needs to be the best on average, i.e.,

$$w^* = \arg\min_w E_{R \sim Bernoulli(p)} ||y - (R * X)w||_2^2 \tag{3.13}$$

In the space given by the orthonormal basis Q^T of the eigendecomposition of the feature covariance matrix $X^TX = Q\Lambda^2Q^T$, the previous equation is equivalent to

$$w^* = \arg\min_w ||y - Xw||_2^2 + \frac{1-p}{p}||\Lambda w||_2^2 \tag{3.14}$$

Here Λ is the diagonal matrix with eigenvalues of $\sqrt{X^TX}$, which represent the standard deviation within dimension of the elements of Q^TX. Thus, it is equivalent to weight decay, adjusted to the magnitude of the dimensions of Q^TX, which by itself is equivalent to the prior normal distribution over weights w with the X^TX covariance matrix.

In the case of logistic regression, and more generally a neural network it is hard to obtain a particular form of Equation (3.14) equivalent to (3.13). Under the assumptions of a normal distribution of weights and neuron input values, the authors of [109] developed the method called *fast dropout*, where they demonstrated how to marginalize out the noise and update the learning algorithms in many cases including neural networks. For example, logistic regression with dropout is approximately equivalent to its maximum likelihood with the random weights $\tilde{w}$ such that $\tilde{w}_i \sim N(\mu_i, \alpha\mu_i^2)$, where $\mu_i = w_i/p$ and $\alpha = (1-p)/p$. They could avoid Monte-Carlo sampling of random noise (standard dropout), which enables a speed up of training process by an order of magnitude. However, as the number of layers increases, the weights and neuron input values might not be distributed normally anymore, so the result cannot be directly applied to deep neural networks. The method yields similar performance to dropout on small networks, but is not used in practice.

Dropout can be also seen as noise added to the input layer [52]. This can be demonstrated by propagating the dropout modifications back to input such that the updated input $\tilde{x}$ causes the same effect as dropout. Thus, it can be viewed as a special way of data augmentation without domain knowledge.

On the test stage the activation values are multiplied by p in order to compensate for the increased amount of non-dropped neurons compared with the training stage. For $d = 0.5$ this method also has a theoretical explanation. If a network has a "softmax" output layer and a hidden dropout layer with N neurons, then this way of testing produces probabilities equal to the geometric

mean of the probabilities produced by each of 2^N networks given by all possible combinations of dropped neurons. Moreover, due to the Jensen inequality the log probability of the correct answer of the averaged model is lower than the mean of the log probabilities produced by individual networks.

Let us consider a multilayer network with a fully connected softmax (2.12) output layer. We denote its input as a vector x of length N, its output as y of the length M, and the matrix of the weights as w with bias b. Let d be the binary dropout mask of the length N, applied to x. In total there are 2^N of them. Then we can compute the geometric mean of the outputs y_i of all 2^N networks, corresponding to different masks d:

$$
\begin{aligned}
P(y_i) = \sqrt[2^N]{\prod_{k=1}^{2^N} softmax(x; w, b, d_k)_i} = \sqrt[2^N]{\prod_{k=1}^{2^N} \frac{exp(xw * d_k + b)_i}{\sum_{j=1}^{M} exp(xw * d_k + b)_j}} = \\
= \frac{1}{C} exp\left(\frac{1}{2^N} \sum_{k=1}^{2^N} (xw * d_k + b)_i\right) = \frac{1}{C} exp\left(\frac{1}{2} xw + b\right)_i = \\
= softmax(\frac{x}{2}; w, b)_i
\end{aligned}
\tag{3.15}
$$

Here C is the normalization constant in the denominator, which is the same for all y_i. Thus, we proved the statement above.

3.8.2 Drawbacks

While dropout works very well on a majority of tasks, it also has limitations. First of all, due to the randomness in architecture it increases convergence time. It requires up to several times lower learning rates to achieve the save variance of the test error as without dropout. Second, on small datasets (< 1000 objects) it does not give any improvement [93]. If the model is large enough, it also remembers all dropout noise, which make it worse. In this case other regularization methods might be more efficient. Third, it has the applicability limit on another side as well. On very large datasets ($> 15M$ objects) the obtained improvement is negligible, so for computational reasons it is recommended to not use it [3].

3.8.3 Further development

In fact, the idea of random temporal neuron removal can be applied to all network-like architectures. For example, in [120] and [81] it was shown, that dropout can be applied to recurrent neural networks (Section 2.2.11). However, temporal connections w_i^2 can not be dropped. Dropout can also be applied to unsupervised models. It was found to be useful in restricted Boltzmann machines [93], and autoencoders [32]. It also caused the appearance of other stochastic training methods with shared information, like stochastic pooling [121].

3.9 Batch Normalization

One of the most recently proposed regularization methods is called *batch normalization* ([43]). It assumes normalization of each layer neurons to have zero mean and unit variance over each training batch. It allows us to use much higher learning rates and saturating nonlinearities (such as sigmoid functions) without a risk of divergence or being stuck in the saturated regime.

Formally, the authors propose to modify each layer values as

$$\hat{y}_i = BN(y_i) = \gamma_i \frac{y_i - \mu_i}{\sqrt{\sigma_i^2 + \epsilon}} + \beta_i, \text{ where } \mu_i = \frac{1}{M}\sum_{j=1}^{M} y_i^j, \text{ and } \sigma_i^2 = \frac{1}{M}\sum_{j=1}^{M}(y_i^j - \mu_i)^2$$

Here γ_i and β_i are two new parameters which need to be trained. They are introduced in order to guarantee that the normalization procedure transforms the values only when it is necessary. Alternatively, the training algorithm can learn them to be $\gamma_i = \sqrt{\sigma_i^2 + \epsilon}$ and $\beta_i = \mu_i$ that makes $\hat{y}_i = y_i$. The gradients for the parameters γ_i and β_i can be easily derived from the definition, and computed by the backpropagation algorithm.

In order to preserve the magnitude of the gradients from the top to the bottom layer, the weight matrices need to have singular values close to 1. It has also be shown to be beneficial for training in [87]. It can be easily shown that the algorithm makes the classifier invariant to scaling of the matrices of weights,

i.e., for any non-zero scalar a

$$BN(y_{i-1} \cdot w_i) = BN(y_{i-1} \cdot (aw_i))$$

Since the singular values are proportional to the matrix scaling, it allows us to meet this requirement. While batch normalization introduces two new parameters, their default values $\gamma = 1$ and $\beta = 0$ effectively move the most probable area of weights to be closer to the origin, thus reducing a chance of the algorithm to stuck in a local minima on the way from the origin to this area.

3.10 Model aggregation

In 3.6 we discussed the method of data augmentation, which is not specific to neural networks but can also be applied to other models. Here we consider another general method of accuracy improvement, called model aggregation, or *ensembling*. The idea is simple: if a model has some random component (for example, the training set), then it is possible to train several models and average their predictions, which are usually better than a single model's predictions. There is a mathematical explanation of this fact.

Let a single regression model $\hat{y}_i$ predict a target y with a normally distributed random noise $\epsilon_i \sim N(0, \sigma^2)$. Moreover, let us assume a constant covariance $E[\epsilon_i \epsilon_j] = C \, \forall \, i \neq j$. Then the averaged prediction $\sum_{i=1}^{N} \hat{y}_i$ has the expected variance

$$\mu^2 = E\left[(\frac{1}{N}\sum_{i=1}^{N}\epsilon_i)^2\right] = \frac{1}{N^2}E\left[\sum_{i=1}^{N}\epsilon_i^2 + \sum_{i\neq j}\epsilon_i\epsilon_j\right] = \frac{1}{N}\sigma^2 + \frac{N-1}{N}C$$

If all models make the same mistakes, i.e., $\sigma^2 = C$, then the average prediction is the same. However, usually models make different mistakes on the same data, so $C < \sigma^2$. If all errors ϵ_i are completely independent, then $C = 0$, so $\mu^2 = \sigma^2/N$. In this case model aggregation is very efficient.

As for the observation that all regularization methods do not decrease the model complexity, the main drawback is increased training time. It is required

K times more time to train K models, than a single one. In fact, it is better when the averaging models are different, because in this case they are more likely to have independent errors.

If a single model is used, ensembling requires some random component. Standard implementations of neural networks already have two of them: random initial weights, and random order of training objects. If on-the-fly data augmentation is used, then the third component is the parameters of the functions $g_i(x, \theta)$. Non-deterministic versions of neural networks have their own random parameters.

Ensembling is reliable regularization method, which can improve any model. For this reason is rarely used in academic research for algorithm comparison, but is very popular in industrial applications, which aim to achieve the best accuracy in all possible ways. For example, the majority of winning entries in Kaggle[1] website competitions apply model averaging.

If the model does not have any random component, it is possible to perform ensembling by training it on a different training sets. If each of them is created by sampling with replacement from the original dataset, and has the same size, it is called *bagging*. Dropout, described in Section 3.8, can be also seen as special way of bagging, where we sample different networks corresponding to different combinations of dropped neurons.

3.11 Tangent propagation algorithm

One of the major problems in image classification is transformation invariance. While a combination of convolutional and pooling layers allows us to achieve invariance to translation, it is not able to deal with rotation and scaling. An interesting method of training a classifier that is robust to a particular transformation was proposed in [89].

As in Section 3.6, let $g(x; \theta)$ be a differentiable invariant transformation function, $g(x, 0) = x$, that preserves the label within a local neighborhood of $\theta = 0$. For simplicity we assume θ to be a scalar. A constant label means constant a

[1] https://www.kaggle.com/

value of a loss function $l(y, f(x))$, so we get the condition on its gradient with respect to θ:

$$\nabla_\theta l(y, f(g(x, \theta)))|_{\theta=0} = 0$$

In other words, the value of the loss function $l(y, f(\tilde{x}))$ at the points $\tilde{x} = g(x; \theta)$ for all small θ should be very close to $l(y, f(x))$.

This is possible to achieve by using an additional regularization term of the minimization function L_{min}:

$$L_{min} = L_{train} + \beta L_{inv} \text{ ,where } L_{inv} = \frac{1}{N} \sum_{i=1}^{N} \tilde{l}(\nabla_\theta l(y, f(g(x, \theta)))|_{\theta=0}),$$

The authors propose to use an MSE loss function $\tilde{l}(x) = ||x||_2^2$. The parameter β controls the strength of regularization.

The described regularizer is not specific for neural networks, but the algorithm of its calculation is specific. Using the condition $g(x, 0) = x$, and the chain rule, we get

$$\nabla_\theta l(y, f(g(x, \theta))_{\theta=0} = \nabla_x l(y, f(x)) \cdot \nabla_\theta x$$

Here $\nabla_\theta x$ is the *tangent vector*, associated with the transformation $g(x; \theta)$. Using the same chain rule, we can represent the backward pass as

$$\tilde{f}(z) = \nabla_x l(y, f(x)) \cdot z = \prod_{i=K+1}^{1} J_i(y_{i-1}) \cdot z,$$

where $y_0 = \tilde{x}$, K is the number of layers, and $J_i(y_{i-1})$ is the Jacobian matrix of a function f_i, computed at its input value y_{i-1}.

The authors propose to compute $\tilde{f}(\nabla_\theta x) = \nabla_\theta l(y, f(g(x, \theta))$ by initializing the network with a tangent vector $\nabla_\theta x^T$ and propagating it through a *linearized* network, i.e., consecutively multiplying it on the transposed Jacobians $J_i^T(y_{i-1})_{i=1,\dots,K+1}$. Indeed,

$$\nabla_\theta x^T \cdot \prod_{i=1}^{K+1} J_i^T(y_{i-1}) = \prod_{i=K+1}^{1} J_i(y_{i-1}) \cdot (\nabla_\theta x) = \nabla_x l(y, f(x)) \cdot \nabla_\theta x = \tilde{f}(\nabla_\theta x)$$

It is easy to notice $\nabla_x l(y, f(x))$ is already computed at the end of the backward pass, so the algorithm can be simplified. We will discuss this in detail in Section 4.3.

In order to update the weights w_i, the gradients $\partial \tilde{l} / \partial w_i$ have to be computed. The authors of [89] just repeat the backpropagation algorithm. They compute the loss function $\tilde{l}(\tilde{f}(\nabla_\theta x)) = (\tilde{f}(\nabla_\theta x))^2$, and perform the backward pass, computing $\partial \tilde{l} / \partial w_i$.

The authors also describe a way to compute tangent vectors $\nabla_\theta x$ for basic image transformation functions. First, the images need to be approximated by a differentiable function $I(x, y)$, that specifies the pixel intensity for any real pair (x, y). This can be done by Gaussian filtering or spline interpolation. Then tangent vectors can be computed by applying corresponding Lie operators to $I(x, y)$. For example, horizontal and vertical invariances are given by operators $\partial/\partial x$ and $\partial/\partial y$, and rotation is given by $y \cdot \partial/\partial x - x \cdot \partial/\partial y$.

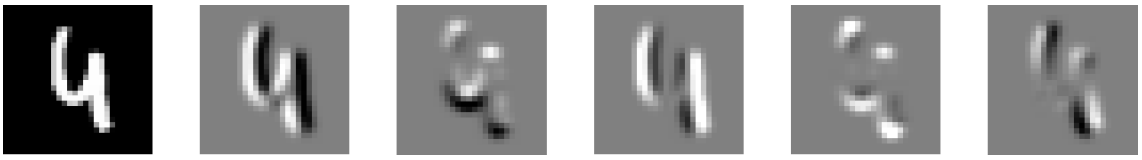


Figure 3.8: From the left to the right: 1) the original image, 2-3) translation tangent vectors, 4-5) scaling tangent vectors, 6) rotation tangent vector.

From the description of tangent propagation algorithm we can see that each invariant transformation function $g(x; \theta)$ requires the same time as the original backpropagation procedure. Thus, in order to achieve invariance to 5 transformations (x and y translations, x and y scaling, and rotation) we need 6 times more time. This is a significant drawback of the algorithm. Moreover, the algorithm requires to compute tangent vectors, which makes its implementation more complicated.

Tangent BP is similar to the Invariant Backpropagation algorithm, presented in Chapter 4. In fact, Invariant BP can be modified to implement Tangent BP more efficiently. We discuss this Section in 4.3.

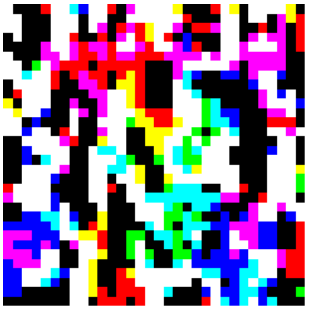
3.12 Adversarial Training

In recent years the quality of image classification has improved a lot. However, there is still a big difference in between people and machine image perception properties. In [95] the authors described an interesting phenomena: it is possible to artificially generate an image indistinguishable from the image of the dataset, such that a trained network's prediction about it is completely wrong. Of course, people never do such kinds of mistakes. These objects were called *adversarial examples*. Formally they can be found as a solution of the minimization problem

$$\tilde{x}^* = \arg\min_{\tilde{x}} l(\tilde{y}, f(\tilde{x})) + c|x - \tilde{x}|_1 \text{ subject to } \tilde{x} \in [0,1]^m$$

Here $l(y, f(x))$ is the loss function of predicting $f(x)$ on an object with a label y. The constant $c > 0$ should be small enough such that the solution $\tilde{x}$ still belongs to the class $\tilde{y}$. Of course, the task is not trivial only when $\tilde{y}$ is different from the label y of the object x, otherwise $\tilde{x} = x$.

(a) x - original image
99.99% - deer, 0.01% - cat



(b) $sign(\nabla_x L)$

(c) $x + sign(\nabla_x L)/256$
3.09% - deer, 96.91% - cat

Figure 3.9: Demonstration of an adversarial example from the CIFAR database (Test set # 27). The correct label is "deer". The class probabilities predicted by a trained network are given.

The authors also discovered that adversarial examples are the same for networks trained with different initialization parameters, and even different training sets. Therefore, they are not just a result of overfitting, but rather the uni-

versal phenomena of neural networks.

From a practical point of view, adversarial examples might be potentially harmful for neural network applications. For example, one might modify a regulated image or video such that it retains its value for people, but fools all security systems that are designed to assess it.

One of the explanations of adversarial examples is given in [33]. The authors argue that the reason is high dimensionality of the input vector, combined with the low flexibility of a classifier on small regions of the input space. Let us consider the gradient of the loss function with respect to the input vector $\nabla_x l(x)$. If ϵ is small, the new object $x^* = x + \epsilon\, sign(\nabla_x l)$ will be close to x, and will have the same label. If x is N-dimensional, then the values of the loss function for the new object is

$$l(x^*) \approx l(x) + \epsilon\, sign(\nabla_x l(x)) \cdot \nabla_x l(x)^T = l(x) + \epsilon ||\nabla_x l(x)||_1 = \epsilon NM,$$

where M is the average magnitude of $|\nabla_x l(x)|$ across all dimensions. The larger is N, the quicker $l(x)$ grows in this direction. In the image classification task N is the number of pixels, which might be very large. At the same time the parameter ϵ might be chosen to be less than $1/255$, which guarantees that the modified image x^* will be perceptually the same as the original image x.

Another comprehensive analysis of adversarial examples is given in [26]. The authors come to the similar conclusion that adversarial examples appear in high dimensional space for the classifiers that are not flexible enough, and therefore appear not only in neural networks. They also give a numerical estimation of robustness to adversarial perturbation for linear and quadratic models, and demonstrate that it is uncorrelated with the robustness to random noise.

In the majority of classification tasks, class probabilities are supposed to be continuous with respect to the input vectors. This means that all objects within a small neighborhood of each training object have the same label as this object. Adversarial examples demonstrate that this knowledge is not automatically learned by a neural network, so it has to be done manually by some regularization method.

One of the straightforward ways to do it is to additionally train a network

on such adversarial examples. While the algorithm of Szedegy ([95]) is quite slow, the algorithm of Goodfellow ([33]) provides a fast way to generate adversarial examples using the derivative at the end of the backward pass $\nabla_x l$. We refer to this algorithm as *Adversarial Training*, or AT. It is also quite similar to the Invariant Backpropagation algorithm, described in Chapter 4, but is developed independently from another perspective. In Section 4.3.2 we provide their comparison, and demonstrate that it can also be sped up to achieve a performance equivalent to IBP.

3.13 Conclusion

In this Chapter we described the most popular regularization methods for neural networks. We demonstrated that many of them can be considered as modifications of the standard L_1/L_2-regularizers with different penalty terms. The convolutional and subsampling layers represent another class of regularizers, which decrease the number of parameters in order to impose particular properties on a classifier. In the end we discussed recently proposed regularization methods: dropout, batch normalization and adversarial training, and provided an explanation from the regularization point of view. The connections between Tangent propagation algorithm, Adversarial Training, and Invariant Backpropagation algorithm are discussed in detail in Chapter 4. The methods of selection of the SVM hyperparameters, including the strength of L_2-regularization (C), are discussed in Chapter 5.

Chapter 4

Invariant Backpropagation

This chapter is based on the following publication: *Invariant backpropagation: how to train a transformation-invariant neural network. Sergey Demyanov, James Bailey, Ramamohanarao Kotagiri, and Christopher Leckie. arXiv preprint arXiv:1502.04434 (2015). Submitted to the International Conference of Learning Representations 2016*

4.1 Introduction

Neural networks are widely used in machine learning. For example, they are showing the best results in image classification [94, 62], image labeling [47] and speech recognition [17]. Deep neural networks applied to large datasets can automatically learn from a huge number of features, that allow them to represent very complex relations between raw input data and output classes. However, it also means that deep neural networks can suffer from overfitting, and different regularization techniques are crucially important for good performance.

It is often the case that there exist a number of variations of a given object that preserve its label. For example, image labels are usually invariant to small variations in their location on the image, size and rotation. In the area of voice recognition the result has to be invariant to the speech tone, speed and accent. Moreover, the predictions should always be robust to random noise. However, this knowledge is not incorporated in the learning process.

In this work we propose a method of achieving local invariance, that **an-**

alytically enforces robustness of predictions to **all types** of variations in the input vector, which are not related with its class. As an extension of the original backpropagation algorithm, it can be applied to all types of neural networks in combination with any other regularization technique. We call this algorithm **invariant backpropagation**, or simply **IBP**.

4.2 Algorithm description

In this section we present our algorithm, and discuss its theoretical and practical aspects. We use the notation introduced in Section 2.2.7.

4.2.1 Algorithm overview

In many classification problems we have a large number of features. Formally it means that the input vectors y_0 come from a high dimensional vector space. In this space every vector can move in a huge number of directions, but most of them should not change the vector's label. The goal of the algorithm is to make a classifier robust to such variations.

Let us consider a K-layer neural network with an input $y_0 = x$, and predictions $y_K = p(x)$. Using the vector of true labels y, we compute the loss function $y_{K+1} = L(p(x))$ [1], and at the end of the backward pass of backpropagation algorithm we obtain the vector of its gradients $dy_0 = \nabla_x L(p(x)) = \prod_{i=K+1}^{1} J_i(y_{i-1})$. This vector defines the direction that changes the loss and therefore changes the predictions. Its length specifies how large this change is. Thus, a smaller vector length corresponds to a more robust the classifier, and vice versa. Let us specify the additional loss function

$$\tilde{L}_p(dy_0) = \frac{1}{p}||dy_0||_p^p, \;\; p > 0 \;\; d\tilde{y}_0 = \frac{\partial \tilde{L}_p(dy_0)}{\partial (dy_0)} \tag{4.1}$$

which is computed at the end of the backward pass. In order to achieve robustness to variations, we need to make it as small as possible.

[1] Since in practice the backpropagation algorithm operates with batches of objects, we denote the loss function by capital L.

Note that $\tilde{L}(dy_0)$ is almost exactly the same as the Frobenius norm of the Jacobian matrix, which is used as a regularization term in contractive autoencoders ([85]). Similar to it, minimization of $\tilde{L}(dy_0)$ encourages the classifier to be invariant to changes of the input vector **in all directions**, not only those that are known to be invariant. At the same time, the minimization of $L(p(x))$ ensures that the predictions change when we move towards the samples of a different class, so the classifier is not invariant in these directions. The combination of these two loss functions aims to ensure good performance.

To optimize $\tilde{L}(dy_0)$, we need to look at the backward pass from another point of view. We may consider that the derivatives dy_K are the first layer of a *reverse* neural network that has dy_0 as its output. Indeed, all transformation functions f_i have reverse pairs $\tilde{f}_i$ that are used to propagate the derivatives (2.10). If we consider these pairs as the original transformation functions, they have their own inverse pairs $\tilde{\tilde{f}}_i$. Later we show that in most cases $\tilde{\tilde{f}}_i = f_i$.

Therefore we consider the derivatives dy_i as activations and the backward pass as a forward pass for the reverse network. As in standard backpropagation, after such a "forward" pass we compute the loss function $\tilde{L}(dy_0)$. The next step is quite natural: we need to initialize the input vector y_0 with the gradients $d\tilde{y}_0 = \nabla_{dy_0}\tilde{L}(dy_0)$ and perform another "backward" pass that has the same direction as the original forward pass. At the same time the derivatives with respect to the weights $d\tilde{w}_i = \nabla_{w_i}\tilde{L}(dy_0)$ must be computed. Fig. 4.1 shows the general scheme of the derivative computation. The top part corresponds to the standard backpropagation procedure.

After performing all three passes we obtain two vectors of derivatives with respect to the weights: dw and $d\tilde{w}$. Now we can define the new rule for the weight updates:

$$w_i \leftarrow w_i - \alpha(dw_i + \beta \cdot d\tilde{w}_i)\ \alpha \geq 0, \beta \geq 0, \tag{4.2}$$

which corresponds to the joint loss function

$$L_{min}(p(x)) = L(p(x)) + \beta\tilde{L}(\nabla_x L(p(x))) \tag{4.3}$$

Here β is the coefficient that controls the strength of regularization, and plays a crucial role in achieving good performance. Note that when $\beta^t = 0$, the algorithm is equivalent to the standard backpropagation.

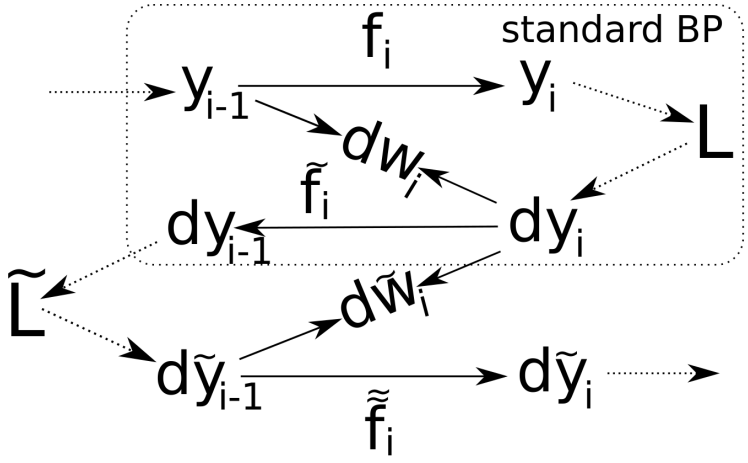


Figure 4.1: The scheme represents three passes if IBP algorithm. Two of them are the parts of standard backpropagation. It also shows which vectors are used for weight derivative computation.

4.2.2 Theoretical details

In the previous section we explained the algorithm conceptually. Here we clarify some details.

First, notice that the forward and backward passes are performed in the same way as in the standard backpropagation algorithm. Then the additional loss function is computed, and its derivatives are used as input for the propagation on the third pass. As it follows from (4.1), for $p = 2$ the gradients are

$$d\tilde{y}_0 = \frac{1}{2}\frac{\partial ||dy_0||_2^2}{\partial (dy_0)} = dy_0,$$

i.e., coincide with the derivatives $dy_0 = \nabla_x L(p(x))$. For $p = 1$, they are the *signs* of dy_0:

$$d\tilde{y}_0 = \frac{\partial ||dy_0||_1}{\partial (dy_0)} = sign(dy_0)$$

In Section 4.2.1 we described double reverse functions $\tilde{\tilde{f}}(d\tilde{y}_{i-1}, w_i)$. Let us additionally introduce functions g_i and their reverse pairs $\tilde{g}_i$ as

$$dw_i = g_i(y_{i-1}, dy_i), \text{ and } d\tilde{w}_i = \tilde{g}_i(d\tilde{y}_{i-1}, dy_i)$$

Now we can formulate the following theorem.

Theorem 1. *Let us assume that f_i is linear, i.e., $f_i(y_{i-1}; w_i) = y_{i-1} \cdot w_i$, where matrix multiplication is used. Then*

1. $\tilde{\tilde{f}}_i = f_i$, *i.e.,* $d\tilde{y}_i = \tilde{\tilde{f}}_i(d\tilde{y}_{i-1}; w) = d\tilde{y}_{i-1} \cdot w_i$,
2. $\tilde{g}_i = g_i$, *i.e.,* $dw_i = y_{i-1}^T \cdot dy_i$, *and* $d\tilde{w}_i = d\tilde{y}_{i-1}^T \cdot dy_i$

Proof. In the case of linear function f_i we know the reverse function $\tilde{f}_i$:

$$dy_{i-1} = \tilde{f}_i(dy_i; w_i) = dy_i \cdot J_i^y(y_{i-1}) = dy_i \cdot w_i^T \tag{4.4}$$

Now let us consider the double reverse functions $\tilde{\tilde{f}}(d\tilde{y}; w)$, such that $d\tilde{y}_i = \tilde{\tilde{f}}_i(d\tilde{y}_{i-1}; w_i)$. Compared with linear f, its reverse function $\tilde{f}$ multiplies its first argument on the transposed parameter. The same is true for the double reverse function $\tilde{\tilde{f}}$ compared with $\tilde{f}$, i.e.:

$$d\tilde{y}_i = \tilde{\tilde{f}}_i(d\tilde{y}_{i-1}, w_i) = d\tilde{y}_{i-1} \cdot (w_i^T)^T = d\tilde{y}_{i-1} \cdot w_i$$

This proves the first statement.

Next, in the case of linear function f_i we also know the function $g_i(y_{i-1}, dy_i)$ which computes the weight derivatives dw_i (2.11):

$$dw_i = g_i(y_{i-1}, dy_i) = y_{i-1}^T \cdot dy_i. \tag{4.5}$$

Let us again consider the backward pass $\tilde{f}_i$ as the forward pass for the reverse net. Since the function $\tilde{f}_i$ is linear, the formula for derivative calculation of reverse net is also (4.5). However, as it follows from (4.4) the reverse net uses the *transposed* matrix of weights for forward propagation, so the result of the derivative calculation is also transposed with respect to the matrix w_i. Also note that

since dy_i acts as activations in the reverse net, we pass it as the first argument, and $d\tilde{y}_{i-1}$ as the second. Therefore,

$$d\tilde{w}_i = g_i(dy_i, d\tilde{y}_{i-1})^T = (dy_i^T \cdot d\tilde{y}_{i-1})^T = d\tilde{y}_{i-1}^T \cdot dy_i, \tag{4.6}$$

and this proves the part 2. □

We thus see that in the case of a linear function f_i, we propagate third pass activations the same way as we do on the first pass, i.e., multiplying them on the same matrix of weights w_i. Note that statement remains true for element-wise multiplication, as it can be considered as matrix multiplication as well. The weight derivatives $d\tilde{w}_i$ are also computed the same way as dw_i in the standard BP algorithm. This theorem allows us to easily implement Invariant BP using the same procedures as for standard BP.

We can also prove another theorem.

Theorem 2. *If the function $f_i(y_{i-1}; w_i)$ has a symmetric Jacobian $J_i(y_{i-1})$, then $\tilde{\tilde{f}}_i(d\tilde{y}_{i-1}; w_i) = \tilde{f}_i(dy_i, w_i)$.*

Proof. Indeed, from (4.4) we can see the argument of the reverse function is multiplied on the transposed matrix of weights. For $\tilde{f}_i(dy_i, w_i)$ the matrix of weights is the Jacobian $J_i(y_{i-1})$. Therefore, if it is symmetric, then $\tilde{\tilde{f}}_i(d\tilde{y}_{i-1}, w_i) = \tilde{f}_i(dy_i, w_i)$. □

This property is useful for implementation of the non-linear functions.

It is easy to compare the computation time for BP and IBP. We know that convolution and matrix multiplication operations occupy almost all the processing time. As we see, IBP needs one more forward pass and one more calculation of weight gradients. If we assume that for each layer the forward pass, backward pass and calculation of derivatives all take approximately the same time, then IBP requires about $2/3 \approx 66\%$ more time to train the network. The experiments have shown that the additional time is about 53%. It is slightly less than the approximated 66%, because both versions contain fixed time procedures such as batch composing, data augmentation, etc.

The summary of the Invariant Backpropagation algorithm is given in Algorithm 4.1.

Algorithm 4.1 Invariant backpropagation: a single batch processing description

1. Perform standard forward and backward passes, and compute the derivatives dw for the main loss function.
2. Perform additional forward pass using the derivatives dy_0 or signs $sign(dy_0)$ as activations. On this pass:
   - do not add biases to activations
   - use backward versions of non-linear functions
   - on max-pooling layers propagate the same positions as on the first pass
3. Compute the derivatives $d\tilde{w}$ for the additional loss function $\tilde{L}$ the same way as dw. Initialize the bias derivatives $d\tilde{w}$ to 0.
4. Update the weights according to the rule 4.2.

4.2.3 Implementation of the particular layer types

A fully connected layer is a standard linear layer, which transforms its input by multiplication on the matrix of weights: $y_i = y_{i-1} \cdot w_i + b_i$, where b_i is the vector of biases. Notice that on the backward pass we do not add any bias to propagate the derivatives, so we do not add it on the third pass as well and do not compute additional bias derivatives. This is the difference between the first and the third passes. If **dropout** is used, the third pass should use the same dropout matrix as used on the first pass.

Non-linear activation functions can be considered as a separate layer, even if they are usually implemented as a part of each layer of the other type. They do not contain weights, so we write just $f(x)$. The most common functions are: (i) sigmoid, $f(x) = 1/(1 + e^{-x})$, (ii) rectified linear unit (*relu*), $f(x) = \max(x, 0)$, and (iii) softmax, $f(x_i) = e^{x_i} / \sum_j e^{x_j}$. All of them are differentiable (except *relu* in 0, but it does not cause uncertainty) and have a symmetric Jacobian matrix, so according to Theorem 2 the third pass is the same the backward pass. For example, in the case of the *relu* function this means that $d\tilde{y}_i = d\tilde{y}_{i-1} * I(y_{i-1} >$

$0)$, where element-wise multiplication is used.

Convolution layers perform 2D filtering of the activation maps with the matrices of weights. Since each element of y_i is a linear combination of elements of y_{i-1}, convolution is also a linear transformation. Linearity immediately gives that $\tilde{\tilde{f}}_i(d\tilde{y}_{i-1}, w_i) = f_i(y_{i-1}, w_i)$ and $d\tilde{w}_i = d\tilde{y}_{i-1}^T \cdot dy_i$. Therefore the third pass of convolutional layer repeats its first pass, i.e., it is performed by convolving $d\tilde{y}_{i-1}$ with the same filters using the same stride and padding. As with the fully connected layers, we do not add biases to the resulting maps and do not compute their derivatives.

The scaling layer aggregates the values over a region to a single value. Typical aggregation functions $f_i(y_{i-1})$ are *mean* and *max*. As it follows from their definition, both of them also perform linear transformations, so $d\tilde{y}_i = f_i(d\tilde{y}_{i-1})$. Notice that in the case of the *max* function it means that on the third pass the same elements of $d\tilde{y}_{i-1}$ should be chosen for propagation to $d\tilde{y}_i$ as on the first pass regardless of what value they have.

4.2.4 Regularization Properties

In the case of L_2 regularizer (4.1), we can derive some interesting theoretical properties. Using the Cauchy-Schwarz inequality, we can obtain

$$||\nabla_x L||_2^2 \leq ||\nabla_x y_K||_2^2 \cdot ||\nabla_{y_K} L||_2^2 \leq ||\nabla_x y_{K-1}||_2^2 \cdot ||\nabla_{y_{K-1}} y_K||_2^2 \cdot ||\nabla_{y_K} L||_2^2$$

The most common loss functions for the predictions $y_K = p(x)$ and true labels y are the squared loss $L(p(x)) = \frac{1}{2}\sum_{i=1}^{M}(p_i(x) - y_i)^2$ and the cross-entropy loss $-\sum_{i=1}^{M} y_i \log p_i(x)$, applied to the softmax output layer $f(p_i) = e^{p_i} / \sum_{j=1}^{M} e^{p_j}$. In the first case we have $\nabla_{y_K} L = y_K - y$, in the second case we can show that $\nabla_{y_{K-1}} L = y_K - y$. Therefore, the strength of L_2-IBP regularization decreases when the predictions y_K approach the true labels y. This property prevents overregularization when the classifier achieves high accuracy. Notice, that if a network has no hidden layers, then $\nabla_x y_{K-2} = w$, i.e., in this case $||\nabla_x L||_2^2$

penalty term can be considered as a weight decay regularizer, multiplied on $y_K - y$.

For the model of a single neuron we can derive another interesting property. In ([7]) it was demonstrated that for a single neuron with the L_2-norm loss function noise injection is equivalent to the weight decay $||w||_2^2$ regularization. In Section 3.7 we show, that if the negative log-loss function is used, noise injection becomes equivalent to the IBP regularizer.

4.3 Alternative approaches

4.3.1 Tangent backpropagation algorithm

Invariant BP is not the first attempt to use derivatives to improve the performance of neural networks. In Section 3.11 we described the Tangent backpropagation algorithm, which is similar to Invariant BP, but trains the classifier to be invariant to particular transformations. We showed that the derivative $\nabla_\theta L(p(g(x,\theta)))$ for the predictions on the modified input $p(g(x,\theta))$ with respect to the transformation parameter θ can be computed more efficiently by multiplying the gradient $\nabla_x L(p(x))$, obtained at the end of the backward pass, on the tangent vector $\nabla_\theta x$. We can demonstrate that Invariant BP can be modified to be a more efficient equivalent of Tangent BP.

In Tangent BP the authors perform an additional iteration of backpropagation through the linearized network, applied to a tangent vector $\nabla_\theta x$. The additional forward pass computes the following values:

$$\tilde{y}_i = \nabla_\theta x^T \cdot \prod_{j=1}^{i} J_j^T$$

If the L_2-norm loss function $\tilde{L}(z) = \frac{1}{2}||z||_2^2$ is used, then $\partial\tilde{L}(z)/\partial z = z$, i.e., the additional gradient is initialized with the value of the loss gradient $\nabla_\theta L = \tilde{y}_{K+1}$.

On the additional backward pass the computed values are therefore

$$d\tilde{y}_{i-1} = \tilde{y}_{K+1} \cdot \prod_{j=K+1}^{i} J_j = \nabla_\theta x^T \cdot \prod_{j=1}^{K+1} J_j^T \cdot \prod_{j=K+1}^{i} J_j$$

According to (2.11), the weight gradients are then

$$\begin{aligned} d\tilde{w}_i = \tilde{y}_{i-1}^T \cdot d\tilde{y}_i &= \left(\nabla_\theta x^T \cdot \prod_{j=1}^{i-1} J_j^T \right)^T \cdot \nabla_\theta x^T \cdot \prod_{j=1}^{K+1} J_j^T \cdot \prod_{j=K+1}^{i+1} J_j = \\ &= \left(\prod_{j=i-1}^{1} J_j \nabla_\theta x \right) \cdot \left(\nabla_\theta x^T \prod_{j=1}^{i-1} J_j^T \right) \cdot J_i^T \cdot \prod_{j=i+1}^{K+1} J_j^T \cdot \prod_{j=K+1}^{i+1} J_j = \\ &= \left(\tilde{y}_{i-1}^T \tilde{y}_{i-1} \right) \cdot J_i^T \cdot \left(dy_i^T dy_i \right) \end{aligned} \tag{4.7}$$

We thus see that in order to compute additional weight derivatives $d\tilde{w}_i$, we need to compute the cumulative Jacobian products from both sides of the network.

In order to obtain the equivalent version of IBP, we need to introduce another layer behind y_0, that would perform multiplication on the tangent vector $\nabla_\theta x$, so $dy_{-1} = dy_0 \cdot \nabla_\theta x$. Its reverse is $d\tilde{y}_0 = d\tilde{y}_{-1} \cdot \nabla_\theta x^T$. The additional L_2-norm loss function $\tilde{L}(dy_{-1})$ is then computed for dy_{-1}. Let us now compute the same gradients $d\tilde{w}_i$ for this IBP modification. As before, the L_2-norm loss initializes the third pass activations with its input dy_{-1}, so

$$d\tilde{y}_i = dy_{-1} \cdot \nabla_\theta x^T \cdot \prod_{j=1}^{i} J_j^T = \prod_{j=K+1}^{1} J_j \cdot \nabla_\theta x \cdot \nabla_\theta x^T \cdot \prod_{j=1}^{i} J_j^T$$

According to (4.6), the gradients are

$$\begin{aligned} d\tilde{w}_i = d\tilde{y}_{i-1}^T \cdot dy_i &= \left(\prod_{j=K+1}^{1} J_j \cdot \nabla_\theta x \cdot \nabla_\theta x^T \cdot \prod_{j=1}^{i-1} J_j^T \right)^T \cdot \prod_{j=K+1}^{i+1} J_j = \\ &= \left(\nabla_\theta x^T \cdot \prod_{j=1}^{i-1} J_j^T \right)^T \cdot \nabla_\theta x^T \cdot \prod_{j=1}^{K+1} J_j^T \cdot \prod_{j=K+1}^{i+1} J_j = \left(\tilde{y}_{i-1}^T \tilde{y}_{i-1} \right) \cdot J_i^T \cdot \left(dy_i^T dy_i \right) \end{aligned} \tag{4.8}$$

Therefore, the weight gradients of both algorithms are the same, so the algo-

rithms are equivalent. However, the IBP modification does not require an extra backward pass. Assuming that the forward, backward, and weight gradient calculation procedures take the same amount of time, Tangent BP is $\approx$ 20% slower. Notice that the same way we can also demonstrate that the modification of Tangent BP, which initializes the second iteration of backpropagation with the identity matrix I instead of transposed tangent vector $\nabla_\theta x^T$, is equivalent to Invariant BP.

In the experiments in Section 4.4 we show that Invariant BP demonstrates better performance than Tangent BP. The reason for this is now more visible. Tangent BP minimizes $\nabla_x L(p(x)) \cdot \nabla_\theta g(x;\theta)$, making a classifier robust only in a particular direction. The vector $\nabla_x L(p(x))$ might be still large. In contrast, Invariant BP minimizes $||\nabla_x L(p(x))||_p^p$, i.e., the vector itself, thus yielding robustness to any direction.

The usage of tangent vectors has other consequences. First, it makes Tangent BP more difficult to implement. To do this, the authors of [89] suggest to obtain a continuous image representation by applying a Gaussian filter, which requires additional preprocessing and one more hyperparameter (filter smoothness). While the basic transformation operators are given by simple Lie operators, other transformations may require additional coding. Second, each tangent vector increases training time, because the standard BP iteration must be repeated for each tangent vector. If we use 5 tangent vectors (2 for translation, 2 for scaling, 1 for rotation), we need about 6 times more time to perform training. In contrast, IBP always requires around 53% more time than standard BP.

4.3.2 Adversarial Training

While the Adversarial Training algorithm 3.12 has been developed from another perspective, it is quite similar to Invariant BP. Both of them use the input gradients $\nabla_x L(p(x))$, obtained at the end of the backward pass, to make a classifier invariant to the changes in the input vector caused by its gradient. However, they differ in a couple of aspects.

For simplicity we assume, that Adversarial Training uses $\nabla_x L$ instead of $sign(\nabla_x L)$ to generate new objects, i.e., $x^* = x + \epsilon \nabla_x L$. For L_1-norm loss func-

tion the results remain the same. In Section 4.3.1 we mentioned that Invariant BP is equivalent to a modification of Tangent BP, which initializes the second iteration with the identity matrix. Since it also requires two iterations of backpropagation as well as Adversarial Training, it is easier to compare Adversarial Training with this Tangent BP modification. In this case we see that the only difference is in the third pass. While Adversarial Training propagates the new objects x^* through the **original** network, Tangent BP propagates the gradients $\nabla_x L$ through the **linearized** network. Adversarial Training uses the same labels y for the new object x^* as for the original object x, so the loss function $L(p(x*))$ is the same. Using Taylor expansion, we can get

$$L(p(x^*)) = L(p(x + \epsilon \nabla_x L(p(x)))) = L(p(x)) + \epsilon ||\nabla_x L(p(x))||_2^2 \tag{4.9}$$

We thus see that the loss function $L(p(x^*))$ can be approximated by the joint loss function (4.3), minimized in IBP, with the equivalent parameters $\beta = \epsilon$. In fact, the authors of [33] propose to minimize

$$L_{min}(p(x)) = (L(p(x)) + L(p(x^*)))/2 = L(p(x)) + \frac{\epsilon}{2}||\nabla_x L(p(x))||_2^2 + o(\epsilon)$$

Easy to notice, that the usage of $L_{min}(p(x))$ instead of $L(p(x^*))$ just scales the hyperparameter ϵ, which needs to be tuned anyway. At the same time, the calculation of gradients $\nabla_{w_i} L(p(x))$ takes computation time. Therefore, the Adversarial Training algorithm can be sped up by avoiding the calculation of $\nabla_{w_i} L(p(x))$, and using only the gradients $\nabla_{w_i} L(p(x^*))$. The parameter ϵ must be 2 times less. Under the same assumptions as for IBP, the performance gain is $\approx 20\%$.

While IBP minimizes only the first derivative, and does not affect higher orders of the derivatives of the loss functions $L(p(x))$ such as curvature, Adversarial Training essentially minimizes **all orders** of the derivatives $\partial^n L(p(x))/\partial^n x$ with the predefined weight coefficients between them. In the case of a highly nonlinear true data distribution $P(y|x)$ this might be a disadvantage. In Section 4.4 we show that none of these algorithms outperform another one in all the cases. Since the sped up version of Adversarial Training takes approximately

the same time as IBP, the choice of regularization depends solely on the data.

4.3.3 Noise injection

Assuming Gaussian noise $\mu \sim N(0, \sigma^2 I)$, such that $E[\mu] = 0$ and $E[\mu^T \mu] = \sigma^2 I$, we can get approximate an arbitrary loss function $L(p(x))$ as

$$E[L(p(x+\mu))] \approx E\left[L(p(x)) + \nabla_x L(p(x))\mu^T + \frac{1}{2}\mu H(x)\mu^T\right] = L(p(x)) + \frac{\sigma^2}{2} Tr(H(x)),$$

where $Tr(H(x))$ is the trace of the Hessian matrix H, consisting of the second derivatives of $L(p(x))$ with respect to the elements of x. Solving the differential equation

$$Tr(H(x)) = \sum_{i=1}^{N} \frac{\partial^2 L}{\partial x_i^2} = \sum_{i=1}^{N} \left(\frac{\partial L}{\partial x_i}\right)^2 = ||\nabla_x L||_2^2,$$

for each x_i independently, we can find the following solution:

$$L = -\left[y \ln |\sum_{i=1}^{N} x_i w_i + b| + (1-y) \ln |1 - \sum_{i=1}^{N} x_i w_i - b|\right],$$

where $y \in \{0, 1\}$ is the class label for the object x. Indeed, assuming $p = \sum_{i=1}^{N} x_i w_i + b$, we obtain the first derivatives:

$$\left(\frac{\partial L}{\partial x_i}\right)^2 = \left[l\frac{\pm w_i}{\pm p} + (1-l)\frac{\mp w_i}{\pm(1-p)}\right]^2 = w_i^2 \left(\frac{p-y}{p(1-p)}\right)^2 = w_i^2 \frac{p^2 - 2py + y^2}{p^2(1-p)^2} \tag{4.10}$$

Now we can compute the second derivatives:

$$\frac{\partial^2 L}{\partial x_i^2} = \frac{\partial}{\partial x_i}\left[w_i \frac{p-y}{p(1-p)}\right] = w_i^2 \frac{p^2 - 2py + y}{p^2(1-p)^2} \tag{4.11}$$

Notice, that the last expression uses y instead of y^2. However if $y \in \{0, 1\}$, then $y = y^2$, so the expressions (4.10) and (4.11) are equal. Therefore, when the negative log-likelihood function L is applied to a single neuron without a non-linear transfer function, the Gaussian noise, added to the input vector x, is equivalent to the IBP regularization term $||\nabla_x L||_2^2$. This result is supported by

the discussion in [26], where the authors show that for the linear classifier the robustness to adversarial examples is bounded from below by the robustness to random noise. However, since $Tr(H(x))$ is only the expected value, the quality of approximation also depends on the number of iterations.

4.4 Experiments

Since the standard backpropagation algorithm is a special case of IBP with the parameter $\beta = 0$ (4.2), we wanted to determine if any $\beta > 0$ gives an improvement with respect to the baseline. As a baseline we considered the versions of standard BP with dropout ratio values 0 and 0.5. Dropout was always applied to the last internal fully connected layer. The best values for β were chosen by cross-validation within the training set. These results are presented in Sections 4.4.2 - 4.4.6. Additionally we present the results of comparison with two other most similar approaches, which target the same goal: Tangent BP (Section 4.4.7) and Adversarial Training (Section 4.4.8). Since the proposed algorithm just trains a neural network in a different way, we do not perform the comparison with other classifiers, such as SVM.

We evaluated our modification on four benchmark datasets for image classification: MNIST, CIFAR-10, CIFAR-100 and SVHN, described in Section 2.5.1. In all experiments we used the following parameters: 1) the batch size 128, 2) initial learning rate $\alpha^0 = 0.1$, 3) exponential decrease of the learning rate, i.e., $\alpha^t = \alpha^{t-1} \cdot \gamma$, 4) each convolutional layer was followed by a scaling layer with *max* aggregation function among the region of size 3×3 and stride 2, 5) *relu* nonlinear functions on the internal layers, 6) final softmax layer combined with the negative log-likelihood loss function.

Each dataset was first normalized to have pixel values within $[0\ 1]$ and then the per-image mean was subtracted from each pixel (except SVHN). The SVHN dataset was normalized by applying local contrast normalization as described in [34]. During training each sample was randomly cropped with the following parameters: 1) cropped image size - 28px, 2) maximum shift from the central position in each dimension - 2 pixels, 3) scale range in each dimension - $[0.71\ 1.4]$,

4) random horizontal reflection for CIFAR-10 and CIFAR-100 datasets, 5) value to use if the cropped image is out of the borders of the original image - 0. Test images were centrally cropped without any random factors. After each epoch the training samples were randomly permuted. The initial weights, cropping parameters and permutations were the same for all values of the parameter β, so β was the only one factor of difference. For each experiment we performed 10 iterations with different initial weights and permutations.

4.4.1 Datasets

Here we observe each of the benchmark datasets, describe the employed neural networks and learning parameters.

MNIST

The MNIST dataset [60] contains handwritten digits, stored as black-and-white images of size 28×28. The total number of classes is 10, one for each digit. The dataset is split into 60k training instances and 10k test instances. We employed a network with two convolutional layers with 32 filters of size 4×4 (padding 0) and 64 filters of size 5×5 (padding 2), respectively, and one internal FC layer of length 256. We trained the classifier for 400 epochs with the coefficient $\gamma = 0.98$, so the final learning rates were $0.98^{400} \approx 0.0003$ of the initial ones. This makes the error variance on the final epochs close to zero.

CIFAR

CIFAR-10 and CIFAR-100 [53] are two other popular benchmark datasets for image classification. They consist of colored 3-channel images of size 32×32 such as cats, dogs, cars, and others. While CIFAR-10 has just 10 broad classes, CIFAR-100 has 100 more specific classes for the same images. For training we used a neural net containing 3 convolutional layers with the filter size 5×5 (paddings 0, 2 and 2), and one internal FC layer of length 256. In this case we trained the classifier for 800 epochs with $\gamma = 0.99$, so $0.99^{800} \approx 0.0003$ the same as for the MNIST dataset.

SVHN

Street View House Numbers (SVHN) is another dataset widely used in image classification. It contains more than $600k$ colored images of digits from

house numbers. Like the CIFAR dataset, these images also have the size $32 \times 32 \times 3$. To obtain the validation set and normalizing the images we followed the procedure described in [34]. Instead of subtracting the per-pixel mean, the authors perform local contrast normalization. We employed the same network structure as for CIFAR, but given that we have 10 times more samples, we added one more FC layer of the same size 256. For the same reason we trained the classifier for only 80 epochs, decreasing the learning rates by 10% after each of them.

4.4.2 Standard BP and Invariand BP

The final results of classification accuracy on the full datasets are summarized in Tables 4.1 and 4.2. We can observe a decrease of the error with respect to both versions of BP with and without dropout for all datasets, except SVHN. The best result is achieved for the combination of both IBP and dropout regularizers. Notice that in all cases the best value of β in a combination with dropout is 10 times lower than the best value without it. Thus, when dropout is used, less additional regularization is required. At the same time, the absolute improvement of IBP remains approximately the same regardless of what dropout rate is used.

Table 4.1: Mean errors for standard BP ($\beta = 0$) and Invariant BP ($\beta > 0$) and the best β on different datasets, **without dropout**.
$\star$ means statistically significant according to the Wilcoxon rank-sum test ($\alpha = 0.05$).

	Standard BP	Invariant BP	best β
MNIST	0.426 ± 0.068	$\mathbf{0.384} \pm 0.031\star$	0.02
CIFAR-10	17.73 ± 0.376	$\mathbf{17.31} \pm 0.247\star$	0.005
CIFAR-100	50.37 ± 0.321	$\mathbf{49.60} \pm 0.437\star$	0.005
SVHN	3.640 ± 0.090	3.589 ± 0.122	0.1

As we see from the tables, an additional fully connected layer on the SVHN dataset led to much larger optimal values of β compared with $CIFAR$. However, even with an additional layer the network did not overfit: the obtained improvement was not statistically significant. Thus, we see that the larger the

Table 4.2: Mean errors for standard BP ($\beta = 0$) and Invariant BP ($\beta > 0$) and the best β on different datasets, **with dropout**.
$\star$ means statistically significant according to the Wilcoxon rank-sum test ($\alpha = 0.05$).

	Standard BP	Invariant BP	best β
MNIST	0.410 ± 0.039	$\mathbf{0.371} \pm 0.036\star$	0.002
CIFAR-10	15.90 ± 0.223	$\mathbf{15.67} \pm 0.198\star$	0.0005
CIFAR-100	46.06 ± 0.330	$\mathbf{45.13} \pm 0.270\star$	0.0005
SVHN	3.570 ± 0.052	3.517 ± 0.065	0.01

dataset is, the less it overfits, and the less improvement we can obtain from regularization. In this case if the dataset is complex enough, we can employ larger networks that can capture these complex relations between input data and output classes. While IBP does not give any improvement of accuracy in this case, it still makes the classifier more robust to noise.

4.4.3 Dataset size and data augmentation

We have also established how the dataset size and data augmentation affects the IBP improvement. We performed these experiments on the MNIST dataset using the same structure as described in Section 4.4.1. The results are summarized in Fig. 4.2. It shows the test errors for the training subsets of different size and the corresponding best values of β. We see that smaller datasets require more regularization (i.e., larger β), and the corresponding absolute accuracy improvement is higher. The relative improvement is also higher: in the case of not augmented datasets it is 27% for 1000 samples and only 12% for $60k$. This result matches with the observation we have made for the SVHN dataset.

Data augmentation improves the accuracy even more than IBP. Since these two methods aim to solve the same problem, they compensate each other, and the improvement of IBP in combination with data augmentation is much less. As can be seen from Fig. 4.2, the optimal value of β is also at least 10 times lower. Thus, IBP is most efficient in the case of small datasets when it is not possible to generate additional samples.

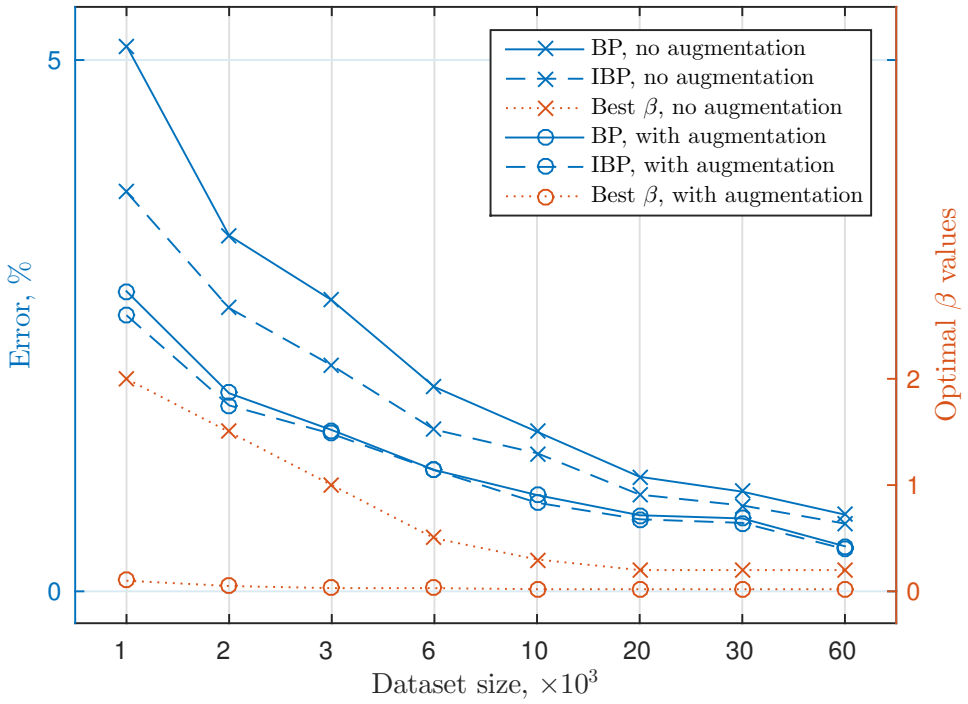
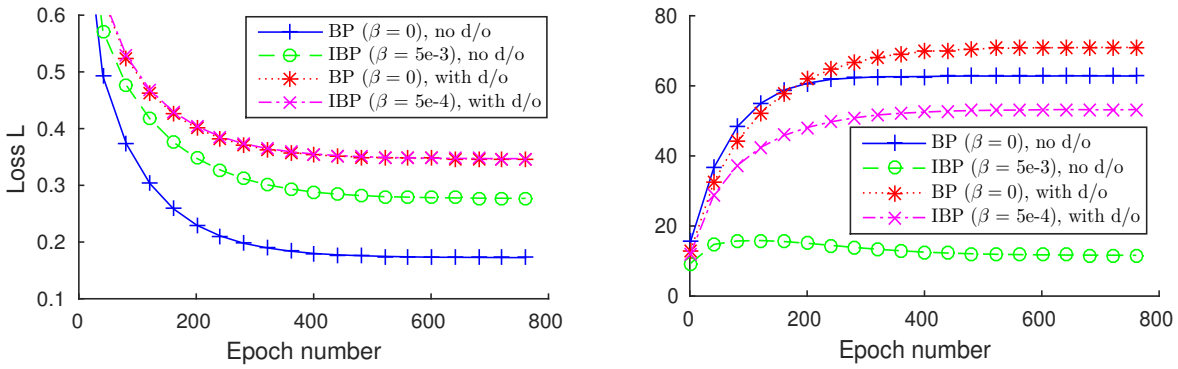


Figure 4.2: Classification errors on the MNIST subsets of different sizes for standard BP and invariant BP with and without data augmentation The corresponding optimal values of β is shown by the dotted lines. Smaller dataset sizes require larger β and get more accuracy improvement.

4.4.4 Loss functions behavior

It is also interesting to look at the plots of the main and additional loss functions. Plots (4.3(a)) and (4.3(b)) demonstrate their curves obtained during the CIFAR-10 training process. We can see that standard backpropagation (blue curves) achieves the lowest value of the main loss L and the highest value the test error. It is a clear sign of overfitting. From (4.3(a)) we can also see that IBP (as well as dropout) acts as a regularizer, increasing L and decreasing the test error. Fig. 4.3(b) confirms that IBP decreases the additional loss $\tilde{L}$, and the larger is β, the lower is $\tilde{L}$.



(a) Main loss function L (b) Additional loss function $\tilde{L}$

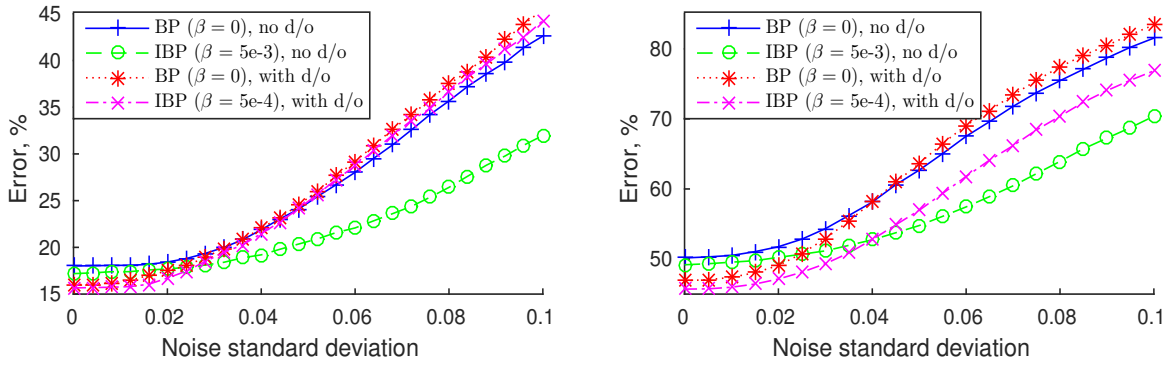
Figure 4.3: The plots of the loss function values on CIFAR-10 dataset for standard BP and Invariant BP with and without dropout. IBP increases the main loss L, but decreases the additional loss $\tilde{L}$. The larger is β, the lower is $\tilde{L}$.

Table 4.3: Computation time of one epoch (seconds)

	BP	IBP	IBP/BP ratio
MNIST	3.80	5.76	1.52
CIFAR-10	5.01	7.46	1.53
CIFAR-100	5.01	7.64	1.53
SVHN	58.6	90.9	1.55

4.4.5 Robustness to noise

Two other plots (4.4(a)) and (4.4(b)) show how the accuracy degrades when we add Gaussian noise to the test set. For this experiment we used the classifiers for CIFAR-10 and CIFAR-100 datasets, obtained in Section 4.4.2. We can clearly see that the green curves that correspond to the classifiers with the largest β have the smallest slope. Pure dropout gives higher accuracy than standard BP, but as we add noise, this classifier degrades faster. At the same time, in the case of CIFAR-100, IBP allows us to preserve its advantage over standard BP when noise is added. These results demonstrate that the theoretically desired properties are observed in practice.



(a) CIFAR-10 dataset

(b) CIFAR-100 dataset

Figure 4.4: The plots of classification errors as functions of noise of the test set for standard BP and invariant BP with and without dropout. In all cases the IBP error increases slower than the corresponding BP error, what demonstrates the robustness of the IBP classifier.

4.4.6 Computation time

As we see from Table 4.3, the additional time is about 53%. It is slightly less than the approximated 66%, because both versions contain fixed time procedures such as batch composing, data augmentation, etc. Thus, we experimentally confirmed that the additionally required time is not more than the theoretical estimate of 66%.

4.4.7 Invariant BP and Tangent BP

We implemented the tangent propagation algorithm and performed experiments on the MNIST and CIFAR-10 datasets. In these experiments we used 5 tangent vectors, corresponding to x and y shifts, x and y scaling and rotation. Since the tangent vectors have to be precomputed in advance, we could not use data augmentation implemented as transformation on the fly, so we did not use data augmentation at all. In order to simplify the design, we did not use dropout as well. First, we estimated the performance of tangent BP on simple neural networks with only fully connected layers. We used 2 internal layers of size 256 for MNIST and 3 layers of the same size for CIFAR. The results are summarized

Table 4.4: Mean errors for standard BP, tangent BP and invariant BP on the neural networks with only fully connected layers

	Standard BP	Tangent BP	Invariant BP
MNIST	2.16 ± 0.11	1.96 ± 0.06	$\mathbf{1.67} \pm 0.07$
CIFAR-10	49.85 ± 0.35	46.29 ± 0.26	$\mathbf{44.06} \pm 0.38$

in Table 4.4. We see that the accuracy is far from the best results, but the relative improvement of the tangent BP is about 2 times less than IBP. Tangent BP also required 3.8 more time for 1 epoch than standard BP.

Next, we performed the experiments on the same network structures that were used to obtain the results from Tables 4.1 and 4.2. We did not use data augmentation and dropout, all parameters were the same. Unfortunately, we could not find any value of $\beta > 0$ that would give a statistically significant improvement of classification accuracy. Thus, we have demonstrated that on these 2 datasets IBP outperforms tangent BP in terms of both accuracy and time.

4.4.8 Invariant BP and Adversarial Training

We have also performed a comparison with Adversarial Training algorithm. The experiments were conducted on the MNIST and CIFAR-10 subsets of size $10k$, using the same network architecture and parameters as for the experiments in Section 4.4.2. We employed the fast version of Adversarial Training, which computes the weight gradients only for the adversarial examples. For both IBP and AT we evaluated two versions (L1 and L2), which use $sign(\nabla_x L)$ and $\nabla_x L$ accordingly for propagation (IBP) and generation (AT). The results are presented in Tables 4.5 and 4.6.

Table 4.5: Mean errors and best parameters for Standard BP, Invariant BP and Adversarial Training on the MNIST dataset. L1/L2 indicates the norm of the additional loss function.

MNIST	Standard BP	IBP-L1	IBP-L2	AT-L1	AT-L2
Mean error	1.21 ± 0.08	1.09 ± 0.11	1.11 ± 0.08	$\mathbf{0.89} \pm 0.07$	0.96 ± 0.05
Best β or ϵ		0.03	0.2	0.05	0.5

Table 4.6: Mean errors and best parameters for Standard BP, Invariant BP and Adversarial Training on the CIFAR-10 dataset. L1/L2 indicates the norm of the additional loss function.

CIFAR-10	Standard BP	IBP-L1	IBP-L2	AT-L1	AT-L2
Mean error	34.7 ± 0.6	**33.1** ± 0.5	33.7 ± 0.3	34.7 ± 0.6	34.6 ± 0.5
Best β or ϵ		0.003	0.02	0.0003	0.002

We see that on the MNIST dataset the AT algorithm demonstrates a significant decrease of test error (27%), while the decrease of IBP is quite modest (10%). However, on the CIFAR-10 dataset the AT algorithm gives no improvement at all, while the IBP can still decrease the error (on 5%). Therefore, it is not possible to make a conclusion that either IBP or AT is better that another one in all cases. The conditions of their applicability remain the area of further research.

At the same time, the results allow to make a conclusion that in the case of image classification problems L_1-norm is more preferable for both IBP and IT.

4.5 Conclusion

We have described an invariant backpropagation algorithm (IBP): an extension of the standard backpropagation algorithm for learning a neural network that is robust to variations in the initial data. Our algorithm requires an additional forward pass for the input vector derivatives and one more calculation of derivatives. Therefore, it is easy for implementation and only requires around 50% more computation time. It can also be applied together with any other methods of regularization. The experiments have demonstrated an improvement of classification accuracy and robustness to noise. The algorithm might be especially useful in the cases of small datasets when additional data generation is not possible. We believe that its properties will make it useful to improve practical usage of neural networks in the areas of image classification, voice recognition and others.

Chapter 5

Tuning of SVM hyperparameters

This chapter is based on the following publication: *AIC and BIC based approaches for SVM parameter value estimation with RBF kernels. Sergey Demyanov, James Bailey, Kotagiri Ramamohanarao and Christopher Leckie. Proceedings of the Fourth Asian Conference on Machine Learning (ACML), pages 97-112, Singapore, 4-6 November, 2012*

5.1 Introduction

Another important problem of neural networks is the search of hyperparameters, such as those that define a network structure. They include the number of layers, their type, the number of neurons and feature maps, the parameters of regularization, such as β, etc. Usually these parameters are chosen by estimating the test error on a validation set or using cross-validation. However, in the second case multiple retraining does not support modification of the structure of a network on the fly. In this chapter we propose a new method of choosing hyperparameters that adjust the model complexity. The method is based on the AIC and BIC information criteria (Section 2.4.2), that find a trade-off between the fitness of a model and the number of parameters. For simplicity we evaluate our method on a simple but powerful classifier - the Support Vector Machine (SVM). We then show how it can be applied to neural networks.

The support vector machine (SVM) is well known and one of the most pop-

ular methods for classification problems. Originally it was a linear classification method, but by using different types of *kernels*, which calculate the dot product in a transformed feature space, it can model nonlinear separating surfaces, extending the utility and power of the method. One of the most popular SVM kernels is the RBF (Radial Basis Function) kernel.

When using this kernel, it is necessary to choose values for the kernel parameter γ and soft margin parameter C. Together, they define the "dimensionality" of the kernel space (we will later make this concept of dimensionality more precise) and thus a wrong choice for them may cause underfitting or overfitting. The right choice of values for these parameters is crucially important for achieving high classification quality.

Apart from estimating the test error on the validation set or using cross-validation, there exist specific methods of model selection that particularly target SVM classifier. For example, the estimation of the test error is given by the radius-margin bound introduced by Vapnik in [101]. Other examples include the trace bound of [2], compression scheme of [27] and sparse margin bound of [37]. Also noteworthy are the compression approach described in [65] and the span estimation introduced in [102]. Work in [65] makes a comparison between methods that follow this second type of approach and argues that the compression approach and span estimation are the best. The idea of span estimation was further developed in [13], where authors provided a gradient descent algorithm for two similar approximations of Leave-One-Out error. It was shown that the proposed method converges to almost optimal parameters for only a few iterations of the algorithm.

In this chapter we propose two new techniques for SVM parameter value estimation, based on the well known AIC and BIC criteria. The techniques differ in the way likelihood is computed. In both cases we estimate the likelihood of a classifier to be Bayesian, i.e., the optimal classifier for the current data distribution. In the first approach we impose some additional assumptions on the distribution, which allow the estimation of the test error in constant time. In the second approach, we directly estimate the probability that at each point the density of the correct class is higher, which is more accurate, but requires more time.

We now present an outline of the rest of the chapter. In the first section we describe the background for SVMs and kernel functions. In Section 5.4 we present general considerations about how AIC and BIC may be applied as part of the search for optimal SVM parameter values for the RBF kernel. This is then followed by derivations of our two likelihood functions. After this, we conduct an experimental study of the performance of our methods and compare them to existing techniques. Finally, we discuss the implications of our results and outline future work.

5.2 Background on Support Vector Machines

The idea behind SVMs is to construct a separating hyperplane that has the largest distance from the closest points of different classes. It has several important properties. First, it requires to solve a quadratic programming problem, which is convex, and therefore can be efficiently computed even for large numbers of points. Second, it is a sparse method, meaning that the location of the hyperplane depends only on a small number of data points, called the *support vectors*. Third, the ability to use kernels provides an elegant way to generalize for when the separating surface is nonlinear. However, the problem of finding the most appropriate kernel and parameter values for the kernel remains an open problem.

Let us consider the classification problem with two classes $Y = \{-1, 1\}$ in a space $X = \mathbb{R}^d$. We construct the classifier

$$a(x) = sign(\langle w, x \rangle) = sign(\sum_{i=1}^{d} w^i x^i),$$

where $x = (x^1, \ldots, x^d)$ is an extended feature vector with 1, and w is the vector of parameters, optimized by the SVM. The equation $\langle w, x \rangle = 0$ describes the separating hyperplane. The original SVM minimization problem is the follow-

ing:

$$\begin{cases} \frac{1}{2C}\langle w, w\rangle + \sum_{i=1}^{N} \xi_i \to \min\limits_{w,\xi} \\ y_i\langle w, x_i\rangle \geq 1 - \xi_i \ \forall i = 1, \ldots, N \\ \xi_i \geq 0 \ \forall i = 1, \ldots, N \end{cases} \tag{5.1}$$

for some constant $C > 0$. The value of this constant is a hyperparameter of the algorithm.

Instead of this, it is much easier to solve a dual problem, which is a well studied quadratic programming problem:

$$\begin{cases} \frac{1}{2}\sum_{i=1}^{N}\sum_{j=1}^{N} \lambda_i\lambda_j y_i y_j \langle x_i, x_j\rangle - \sum_{i=1}^{N} \lambda_i \to \min\limits_{\lambda}; \\ 0 \leq \lambda_i \leq C \ \forall i = 1, \ldots, N; \\ \sum_{i=1}^{N} \lambda_i y_i = 0. \end{cases} \tag{5.2}$$

When this problem is solved, w is calculated as

$$w = \sum_{i=1}^{N} \lambda_i y_i x_i$$

Here $\lambda_i \geq 0$ only for points that have a distance from the hyperplane $y_i\langle w, x_i\rangle \leq 1$. These points are called *support vectors*. Moreover, $\lambda_i < C$ only for *margin support vectors*, i.e., for points having the distance from the hyperplane $d_i = y_i\langle w, x_i\rangle = 1$. For all other support vectors $\lambda_i = C$ and $d_i < 1$. Notice that points having $\lambda_i = 0$ do not influence the solution (which is why the algorithm has a "sparsity" property).

Another property that makes SVMs popular is the possibility to use kernels. If the dataset is poorly separated in the original feature space, it may be the case that it is better separated in some space H with more dimensions. Consider some function $\phi(x) : X \to H$, which transforms original vectors into such a space. The optimization problem does not depend directly on the features, only on dot products, so the only difference compared to the original SVM formula-

tion is that now we need to use the dot product in the new space $\langle \phi(x), \phi(x') \rangle$.

A function $K(x, x')$ is called a kernel if it can be represented in the form $\langle \phi(x), \phi(x') \rangle$ for some function $\phi : X \to H$. Kernels can be very different, which leads to a significant variety of possible algorithms. In all cases we simply substitute the original dot product by its analogue given by the kernel function, so the classification function is

$$a(x) = sign\left(\langle w, x \rangle\right) = sign\left(\sum_{i=1}^{N} \lambda_i y_i K(x, x_i)\right) \tag{5.3}$$

The RBF and polynomial kernels are the most popular classical SVM kernels. The first one is given by the formula

$$K(x, y) = \langle \phi(x), \phi(y) \rangle = \exp(-\gamma ||x - y||^2)$$

for some $\gamma > 0$, which regulates the classifier flexibility. Although many kernels have been proposed, no single kernel is the best for all problems. When using very small values of γ in the RBF kernel, the dot product for any two points is close to 1, so its prediction does not depend much on a point location. This is the situation of underfitting. At the same time, very large values of γ discourage the contributions from the points that are not the closest, making a classifier work as a 1-nearest neighbor. This is a situation of overfitting. Thus, there is the challenge to select the best values of γ (and C) for the RBF kernel.

5.3 Dimensionality estimation

First, in order to employ the AIC and BIC criteria, we need to identify the dimensionality of the parameter space $|\theta_i|$. In the case of an SVM, which is a linear classifier in the transformed feature space $\phi(x)$, extended by a constant 1, it is equal to the dimensionality of the space d_i. We can show that **the value of d_i is equal to the number of margin support vectors**.

First, we may exclude non-support vectors from consideration, because they do not affect the position of the hyperplane. Next, we have N_{SV}^m margin support vectors (i.e., $0 < \lambda_i < C$), and each of them has a distance from the hyperplane

$d_i = y_i\langle w, x_i\rangle = 1$. Therefore, the system of linear equations

$$\langle w, x_i\rangle = y_i,\ i = 1, \ldots, N_{SV}^m$$

has at least one solution with respect to w. For randomly sampled pairs (x, y) it is possible, only if $d_i \geq N_{SV}^m$.

From another point of view, the SVM optimization problem can be considered as a problem of linear programming for the variables ξ_i with the inequality restriction on $||w||$. Within this restriction, the problem can be solved using the Simplex method. It is known that the solution of the simplex method is necessarily a vertex of a polytope, i.e., the point with d_i active restrictions. In the case of an SVM the restrictions are

$$\xi_i \leq 1 - y_i\langle w, x_i\rangle,$$

which are active only for margin support vectors. Therefore, if there are N_{SV}^m of them, the dimensionality of the space is also N_{SV}^m.

For large values of γ (corresponding to overfitting), the dimensionality $|\theta_i|$ is often equal to N and therefore the denominator in the AIC formula (2.25) is equal to -1. In this situation we considered the value of AIC to be ∞, so it was higher than all other non-infinite AIC values. Since we were searching for the parameter value pair that had achieved the minimum for the AIC, such γ and C values could not be the solution.

5.4 AIC and BIC for the best parameter search

For our purpose we employ the AIC and BIC criteria, described in Section 2.4.2. They contain a likelihood term $L(M_i|Z)$. Next, we offer some perspectives of how to compute this likelihood for the purpose of optimal model complexity search. In the case of an SVM with RBF kernel it assumes the search for the best values of γ and C, which identify an individual model M_i.

Let some p.d.f. be a true distribution for a dataset. Each kernel function transforms it to a new kernel p.d.f. in the transformed feature space. This trans-

formed p.d.f implies a corresponding (optimal) Bayes classifier in this space $a(x) = \arg\max_y P(y|x)$, i.e., the one with the lowest expected value of the test error. Therefore, in order to find the best model we should compute the likelihood that the SVM hyperplane coincides with the Bayes separation boundary. We can notice that in the case of underfitting we can observe a large connected regions of misclassified objects. Therefore, one way is to directly estimate if the correct class y_j indeed has a higher probability $P(y_j|x) > 1/2$ for each point x of a subspace classified as y_j. This is the idea of our first approach. Another way is to make an additional assumption about the distribution of misclassified objects of a Bayesian classifier, which makes the likelihood equal to the SVM objective function. This makes it possible to get the value of the likelihood for free, once the SVM classifier is found.

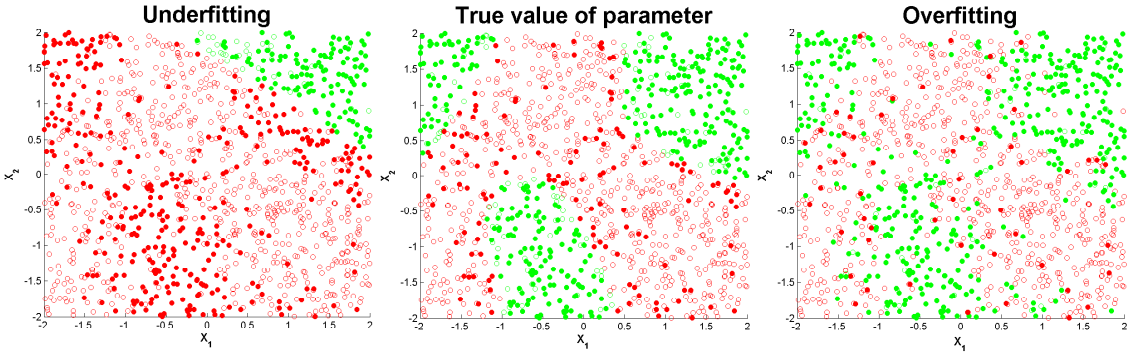


Figure 5.1: The solid dots represent the misclassified objects. On the left picture, which corresponds to underfitting, the large areas of misclassified objects (red solid dots) are visible.

5.4.1 Approach 1: Margin-based approach

We first describe a method of fast likelihood computation. The idea of the method is to use the minimum SVM objective function as the likelihood of a model, required for the AIC/BIC formulas. This can be done without any further considerations. However, it is useful to understand which model has the same likelihood estimation as SVM.

There exist methods to construct such probability distributions. For instance, [101] introduced the third "don't know" class to avoid issues with normaliza-

tion. In [35], probability intervals were used to interpret SVM outputs and thus the SVM solution is approximated by the negative log-likelihood. Work in [82] used a simple sigmoid function for the transformation of outputs to posterior probabilities.

Franc, Zien and Schölkopf in [29] showed that the joint p.d.f. of an observation x and its class y

$$p(x, y|w) = Z(||w||) \cdot \exp(-l(d)) \cdot h(\phi(x))$$

with a value $||w||$ obtained from the SVM solution that satisfies this condition. Here w is a normal vector to the SVM separating hyperplane in the kernel feature space, $l(d) = max(1 - d, 0)$ is the hinge loss function, $d = y\langle w, x\rangle$ is the distance from the hyperplane, $Z(||w||)$ is a normalization constant, and $h(x)$ is some positive integrable function, such that $h(x_1) = h(x_2)\ \forall x_1, x_2 : ||x_1|| = ||x_2||$. For equal misclassification costs, the SVM hyperplane also coincides with the plugged-in Bayes classifier of this distribution. This distribution is thus the additional assumption, that enables the use of the minimum of the SVM objective as a likelihood function. The negative log-likelihood of this distribution is

$$-\sum_{i=1}^{N} \log p(x_i, y_i|w) = \sum_{i=1}^{N} l(d_i) - \sum_{i=1}^{N} \log(h(\phi(x_i))) - N \log Z(||w||) \qquad (5.4)$$

Since $h(\phi(x_i))$ does not depend on w, this term can be excluded from the likelihood. Therefore, the ML problem is equivalent to

$$\arg\min_w [-\log L(w|X, Y)] = \arg\min_w \left[-N \log(Z(||w||) + \sum_{i=1}^{N} l(d_i) \right]$$

Assuming

$$Z(||w||) = exp\left(-\frac{||w||}{2CN}\right),$$

we obtain the definition of the SVM objective function (Equation (5.1)). We can also notice that as in the SVM objective, the points with $d_i > 1$ do not influence the SVM solution, which means the equivalence of the ML solutions for the

general distribution $p(x, y|w)$ and the conditional distribution $p(x, y|d \leq 1, w)$.

Since the negative log-likelihood value coincides with the minimum of the SVM objective function, the method does not require any additional calculations.

5.4.2 Approach 2: Density-based approach

Next we derive the likelihood function that directly estimates the probability of the SVM hyperplane to be a Bayes classifier $a(x) = \arg\max_y p(y|x)$, i.e., the best possible classifier for a particular data distribution $p(x, y)$. This can be done by estimating the probability that at every point of a hyperspace classified as y_i, the probability to find a y_i class object is higher.

Let us consider a binary classification problem with $y_i \in \{-1, 1\}$. Then we can write that

$$p(y_i|x_i) + p(-y_i|x_i) = 1, \forall x_i \in X$$

Considering only the points of the training set, and assuming the independence of their probabilities $p(y|x)$, we can compute the required likelihood as

$$L(w|X, Y) = \prod_i [p(y_i|x_i) > p(-y_i|x_i)] = \prod_i \left[p(y_i|x_i) > \frac{1}{2}\right]$$

Let us fix some point x_i of the training set and a sphere of radius r around it. We denote the number of y_i and $-y_i$ class objects inside this sphere as $n^r_+(x_i)$ and $n^r_-(x_i)$. Assuming that the probabilities $p(y_i|x_i)$ and $p(-y_i|x_i)$ in this sphere are constant, we can consider that $n^r_+(x_i)$ is the number of successes of sampling from the Binomial distribution with the success probability $p(y_i|x_i)$ and $n^r_+(x_i) + n^r_-(x_i)$ number of attempts. Therefore, the probability of such outcome is

$$q(p) = Bi(n^r_+(x_i); n^r_+(x_i) + n^r_-(x_i), p(y_i|x_i))$$

Given the certain values of $n^r_+(x_i)$ and $n^r_-(x_i)$, we are interested in the probability that $p(y_i|x_i) > 1/2$. This can be computed as the ratio of summarized probabilities $q(p)$ for $p > 1/2$, and summarized probabilities $q(p)$ for the whole

possible range $p \in [0, 1]$. Formally, we are interested in

$$Pr\left[p > \frac{1}{2}\right] = \frac{\int_{1/2}^{1} Bi(n_+^r; n_+^r + n_-^r, p)dp}{\int_0^1 Bi(n_+^r; n_+^r + n_-^r, p)dp} = \frac{\int_0^{1/2} p^{n_-^r}(1-q)^{n_+^r}}{\int_0^1 p^{n_-^r}(1-q)^{n_+^r}} = \\ = \frac{B(1/2; n_-^r + 1, n_+^r + 1)}{B(n_-^r + 1, n_+^r + 1)} = I_{1/2}(n_-^r + 1, n_+^r + 1) \tag{5.5}$$

Here $B(a, b)$ is the Beta function and $B(x; a, b)$ is an *incomplete* Beta function. Their ratio $I_x(a, b)$ is called a *regularized* incomplete Beta function, and can be efficiently computed [19, 20].

As for our previous approach, we require that non-support vectors have no influence on the likelihood function. This means that for all non-support vectors $\Pr[p > 1/2] = 1$. In our experiments we have assumed that this is also true for all correctly classified points. Thus, the only contributing points are the non-margin support vectors that have been misclassified.

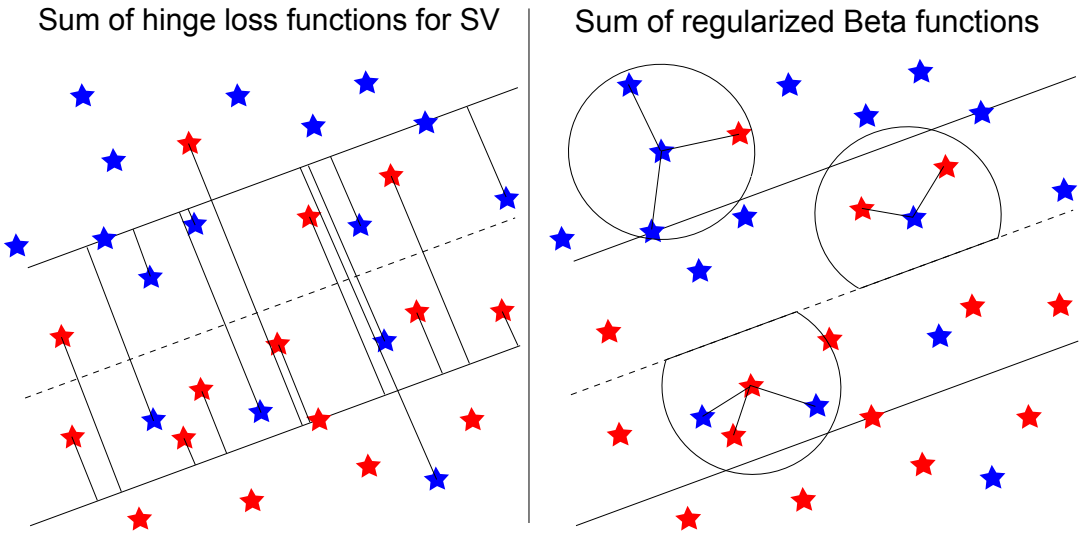
The algorithm also requires the radius r to be chosen. For each misclassified support vector we choose the radius, which yields the **lowest** probability for $p(y_i|x_i)$. Therefore, this way we compute the lower bound for the likelihood function $L(w|X, Y)$. This approach allows us to capture large connected regions of misclassified objects, which is a characteristic of underfitting. If r is larger than the distance to the hyperplane d_i, we do not count the points from another side, because they are expected to have different value of $p(y_i|x_i)$. The algorithm outline is given in Algorithm 5.1.

The likelihood function, computed this way, cannot be differentiated, but its expectation with respect to $\{X, Y\}$ for continuous $p(x, y)$ is also continuous. Therefore the determinant of the Fisher information matrix is non-zero and usage of the AIC/BIC formulas is legitimate. This method requires calculation of the distance between all pairs of points in the dataset. Using the kernel function, it may be computed using Equation (5.6), which requires $O(N^2)$ operations.

$$d(x_i, x_j) = \sqrt{K(x_1 - x_2, x_1 - x_2)} = \sqrt{K(x_1, x_1) - 2K(x_1, x_2) + K(x_2, x_2)} \tag{5.6}$$

When choosing the best radius r, it is sufficient to consider only the values of

the distances between points, because only at these values the likelihood function may change. In order to do this, we need to sort all distances between non-margin support vectors and all other points ($\leq N_{SV}^{nm} N$ values), which requires $O(N_{SV}^{nm} N \log N)$ operations in total. Next, we iteratively take each distance as a radius r and recalculate the probability $\Pr[p(y_i|x_i) > 1/2]$ for this value of radius. It takes $O(N_{SV} N)$ operations, one for each radius. For small γ the number of non-margin support vectors N_{SV}^{nm} might be compared with N, so the worst case performance is $O(N^2 \log N)$. In practice, the algorithm works much faster. For a relatively small dataset (≤ 1500 points) the algorithm takes much less time than the SVM training procedure.



(a) Margin based approach: SVM objective - sum of hinge losses and regulariser.

(b) Density based approach: estimation of class densities at the support vector locations.

Figure 5.2: Visualisation of both approaches for likelihood computation.

5.5 Experiments

This section provides the details of the experimental evaluation of the two proposed approaches.

Work by [65] described a model selection approach based on the compres-

sion quality for different values of γ and C. They showed how to compress the SVM classification results for all points. The main idea was to code the position of the hyperplane with the required accuracy and then add the information about misclassified points. The values of parameters, which yield the highest compression rate, are then chosen. They made a comparison of their compression schemes with different error generalization bounds and demonstrated that the best quality among all tested methods was achieved for their $C2$ and $C3$ compression codes, and span estimation of Leave-One-Out (LOO) cross-validation error, introduced in [102].

Our experiments followed the same design as described in [65]. Training and test subsets were normalized according to the maximum and minimum values of the training set. Instead of fixed sizes $m = 100$ and $m = 500$ we performed computations for all values from 100 to 1500 (when the dataset was large enough), with step size 100. For each dataset, each training set size and each experiment we calculated the test error for the classifier learned with the parameters predicted by a given algorithm. An algorithm curve represents an average test error for 30 different experiments (splits on the training and test parts). Two plots for each dataset are presented, one for each likelihood function. We used the following ranges of parameters: $C \in [10^0, \ldots, 10^5]$, $\gamma \in [10^{-3}, \ldots, 10^3]$. We tested our approach on twelve well-known datasets from the UCI database. Since the dataset "usdigits-0" had ten output labels, we considered the first digit as one class and all others as another class. All experiments were conducted using the LIBSVM software [12].

We compared our two approaches against the $C3$ compression code as a baseline, because it involves eigenvalues only for the restricted kernel matrix $K(x_i, x_j)$, where x_i, x_j are the support vectors, and therefore it has low computation time. However, it requires $O(N_{SV}^3) = O(N^3)$ for large γ, which can be prohibitive even for medium size datasets. Another computationally expensive step is the calculation of the radius of the smallest sphere around the support vectors. This is a quadratic programming problem, but it may be approximated by the maximum distance to their center. This C3 baseline technique is denoted as "Compression".

We did not compare against the span estimation method in our experiments,

Table 5.1: List of datasets

dataset	features	size	positive	negative
diabetes	8	768	268	500
german	20	1000	300	700
image	18	2086	1188	898
banana	2	5300	2376	2924
ringnorm	20	7400	3664	3736
twonorm	20	7400	3703	3697
splice	60	2991	1344	1647
waveform	21	5000	1647	3353
usdigits-0	256	11000	1100	9900
abalone	8	4177	2081	2096
titanic	6	2201	711	1490
thyroid	21	7200	534	6666

since its computation time was prohibitive. Instead, we compared against another baseline of a simple "Hold out" validation: each training set was split into two subsets (75% and 25%), which were used as learning and validation sets, and the best pair of parameters γ and C was the pair that gave the minimum error on this small validation set. This simple method requires only one training procedure and therefore has similar running time to our approaches. This second baseline technique is denoted as "Validation".

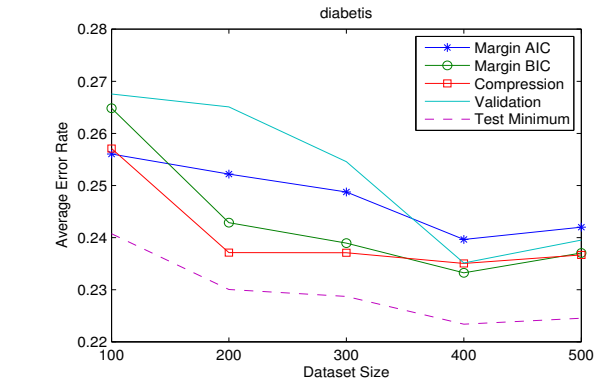
In the plots, the "Test Minimum" curve is an average of the 30 test error minima - the best (and lowest) possible result. We present the results for variations of our methods corresponding to combinations of i) our two different likelihood functions (Margin and Density) and ii) the two different complexity penalty terms (AIC and BIC). Our four methods are thus termed Density-AIC, Density-BIC, Margin-AIC and Margin-BIC.

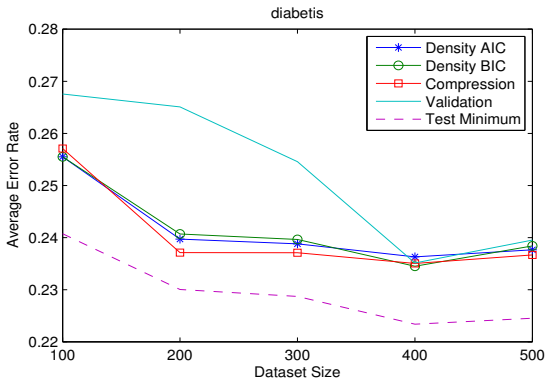
We also assessed the statistical significance of the differences in accuracy between each pair of algorithms. In particular, for each dataset and each algorithm, 90 different error values were selected: 30 for each of the dataset sizes 1300, 1400 and 1500. The 90 values for the first algorithm were then compared against the 90 values for the second algorithm using a left-tail t-statistic and a p-value was obtained. For the smaller datasets, different dataset sizes were used: 300-500 for "diabetes" and 500-700 for "german". The p-value was considered

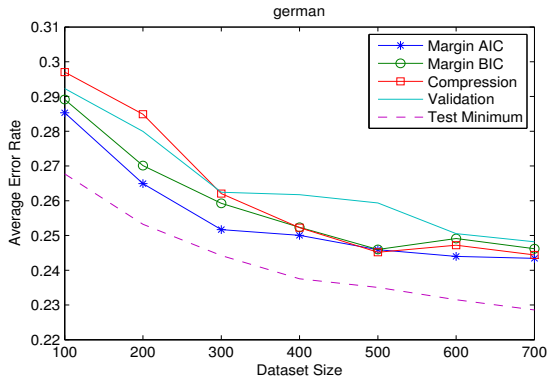
to be significant using a threshold of 0.05. Table 5.2 shows the results.

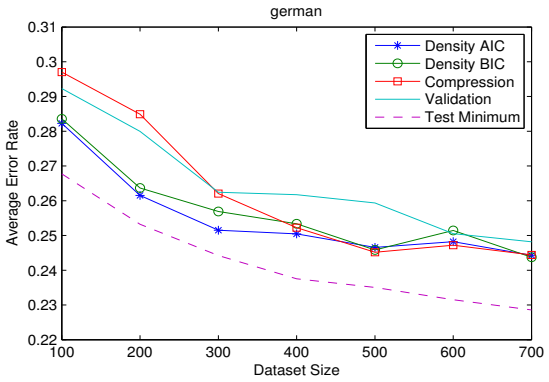
Table 5.2: Statistical significance results for pairs of algorithms. The value of cell (i, j) corresponds to the number of datasets where algorithm i is statistically significantly better than algorithm j.

	MAIC	MBIC	DAIC	DBIC	Compression	Validation
MAIC	0	4	1	8	3	2
MBIC	4	0	3	8	4	4
DAIC	5	6	0	8	7	7
DBIC	3	1	1	0	2	4
Compression	4	3	1	7	0	6
Validation	6	6	4	7	5	0


diabetis
Margin AIC
Margin BIC
Compression
Validation
Test Minimum
Average Error Rate
Dataset Size


diabetis
Density AIC
Density BIC
Compression
Validation
Test Minimum
Average Error Rate
Dataset Size


german
Margin AIC
Margin BIC
Compression
Validation
Test Minimum
Average Error Rate
Dataset Size


german
Density AIC
Density BIC
Compression
Validation
Test Minimum
Average Error Rate
Dataset Size

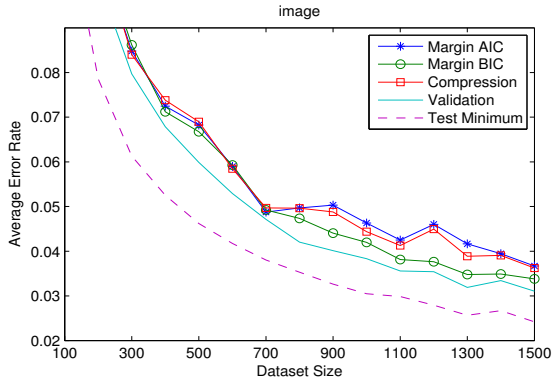


image
Margin AIC
Margin BIC
Compression
Validation
Test Minimum
Average Error Rate
Dataset Size

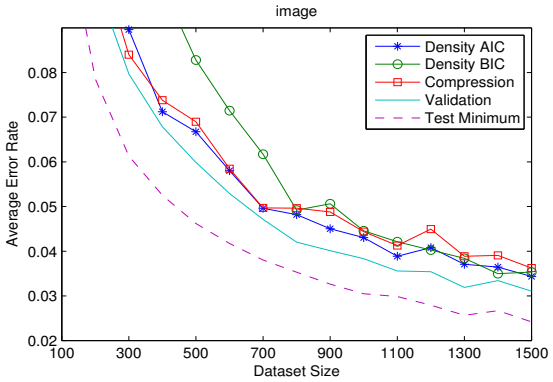
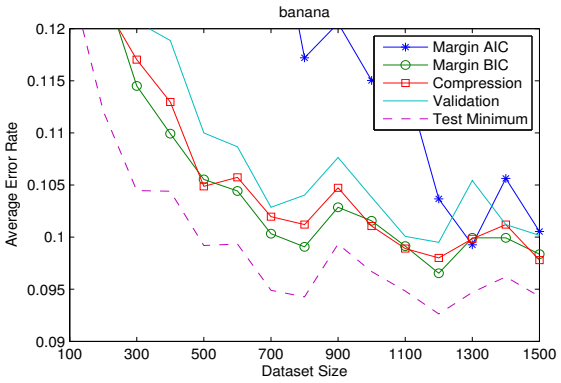
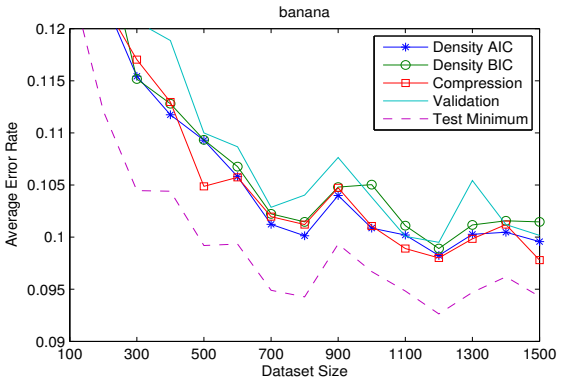
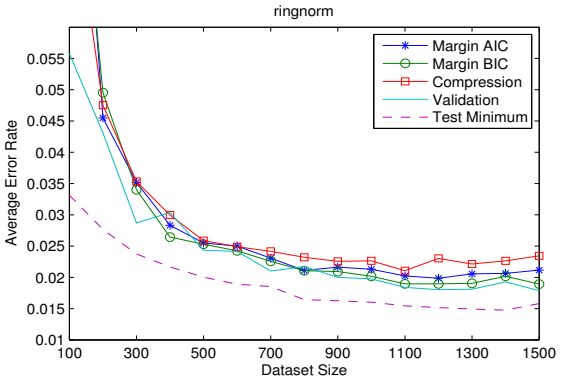
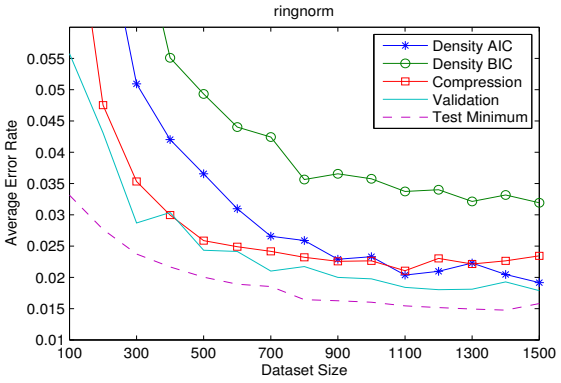


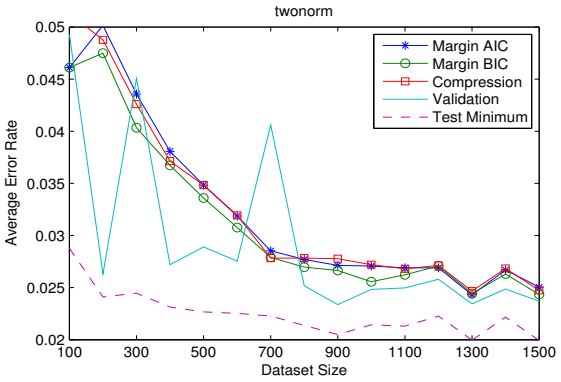
image
Density AIC
Density BIC
Compression
Validation
Test Minimum
Average Error Rate
Dataset Size

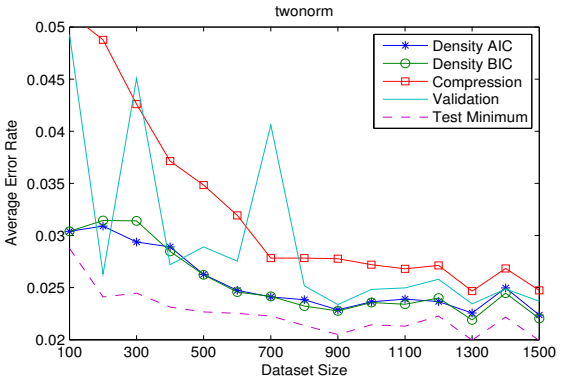

banana
Margin AIC
Margin BIC
Compression
Validation
Test Minimum
Average Error Rate
Dataset Size

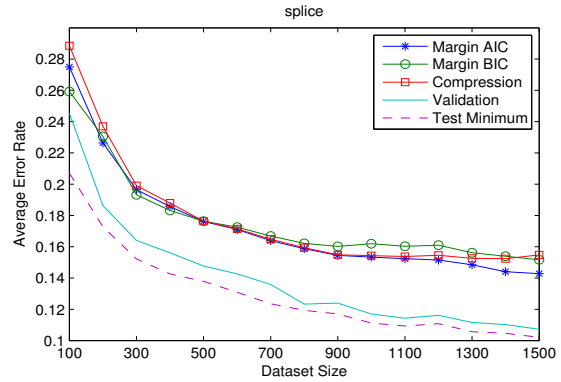

banana
Density AIC
Density BIC
Compression
Validation
Test Minimum
Average Error Rate
Dataset Size

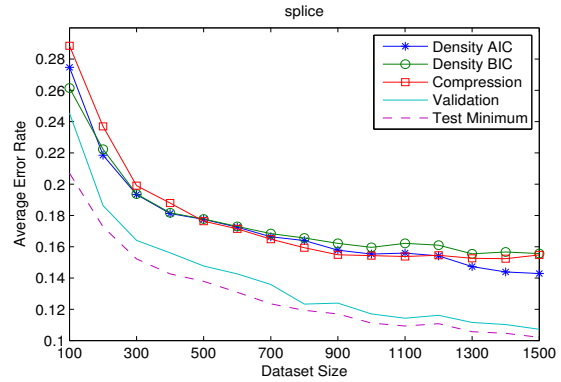

ringnorm
Margin AIC
Margin BIC
Compression
Validation
Test Minimum
Average Error Rate
Dataset Size

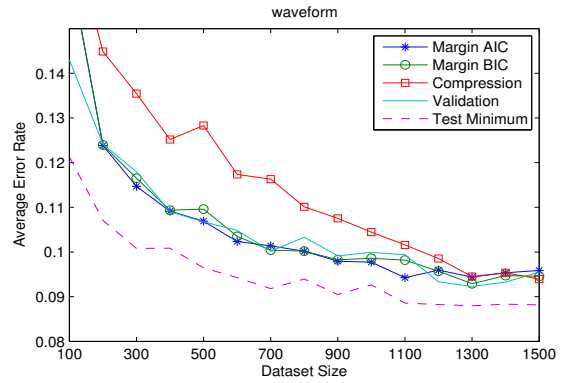

ringnorm
Density AIC
Density BIC
Compression
Validation
Test Minimum
Average Error Rate
Dataset Size

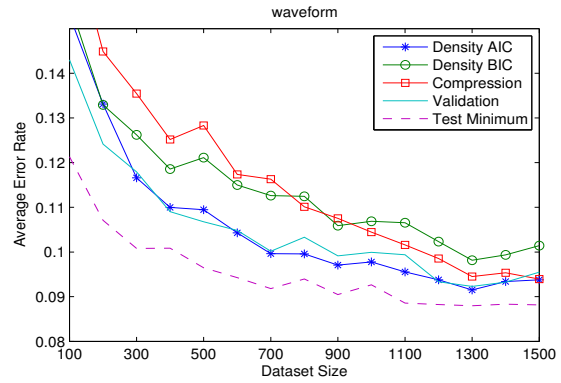

twonorm
Margin AIC
Margin BIC
Compression
Validation
Test Minimum
Average Error Rate
Dataset Size

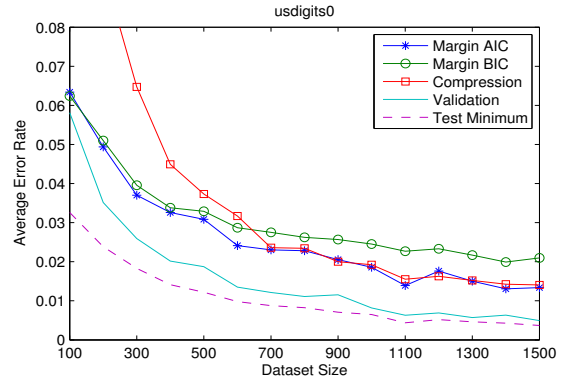

twonorm
Density AIC
Density BIC
Compression
Validation
Test Minimum
Average Error Rate
Dataset Size

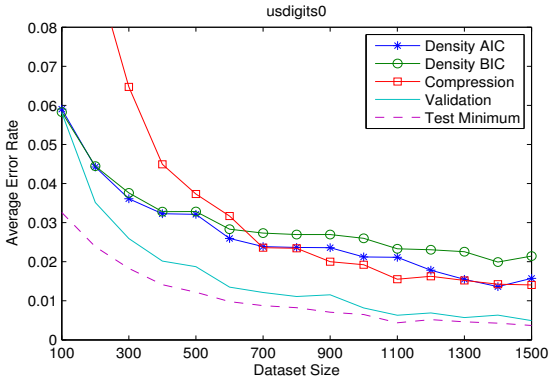

splice
Margin AIC
Margin BIC
Compression
Validation
Test Minimum
Average Error Rate
Dataset Size

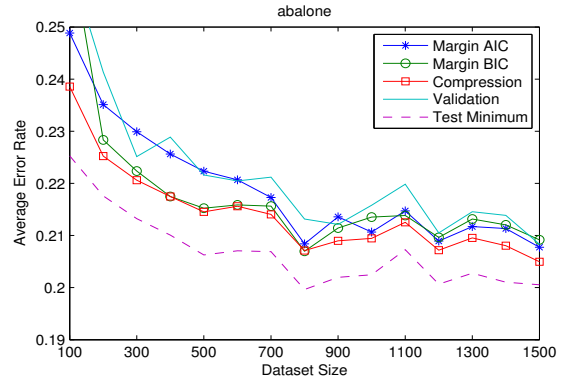

splice
Density AIC
Density BIC
Compression
Validation
Test Minimum
Average Error Rate
Dataset Size

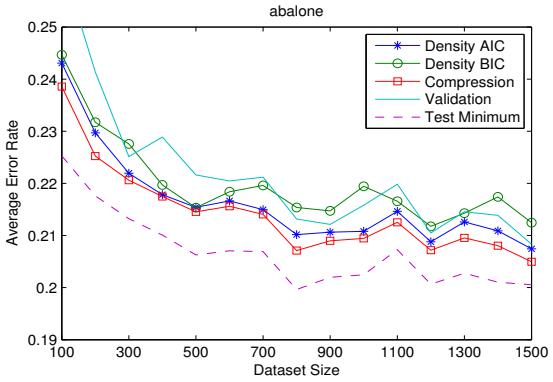

waveform
Margin AIC
Margin BIC
Compression
Validation
Test Minimum
Average Error Rate
Dataset Size

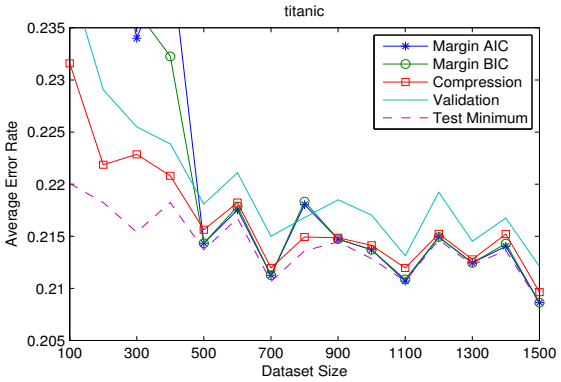

waveform
Density AIC
Density BIC
Compression
Validation
Test Minimum
Average Error Rate
Dataset Size

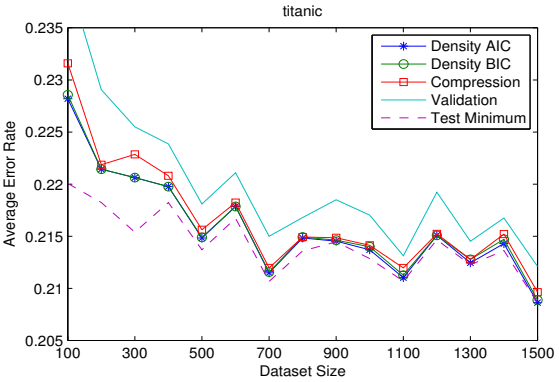

usdigits0
Margin AIC
Margin BIC
Compression
Validation
Test Minimum
Average Error Rate
Dataset Size

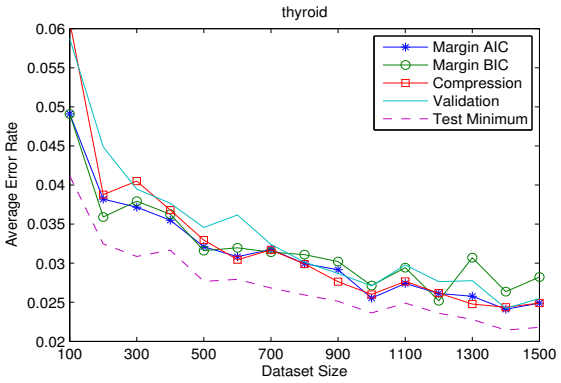

usdigits0
Density AIC
Density BIC
Compression
Validation
Test Minimum
Average Error Rate
Dataset Size

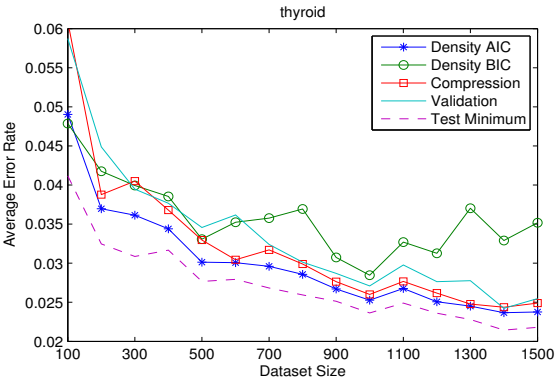

abalone
Margin AIC
Margin BIC
Compression
Validation
Test Minimum
Average Error Rate
Dataset Size


abalone
Density AIC
Density BIC
Compression
Validation
Test Minimum
Average Error Rate
Dataset Size


titanic
Margin AIC
Margin BIC
Compression
Validation
Test Minimum
Average Error Rate
Dataset Size


titanic
Density AIC
Density BIC
Compression
Validation
Test Minimum
Average Error Rate
Dataset Size


thyroid
Margin AIC
Margin BIC
Compression
Validation
Test Minimum
Average Error Rate
Dataset Size

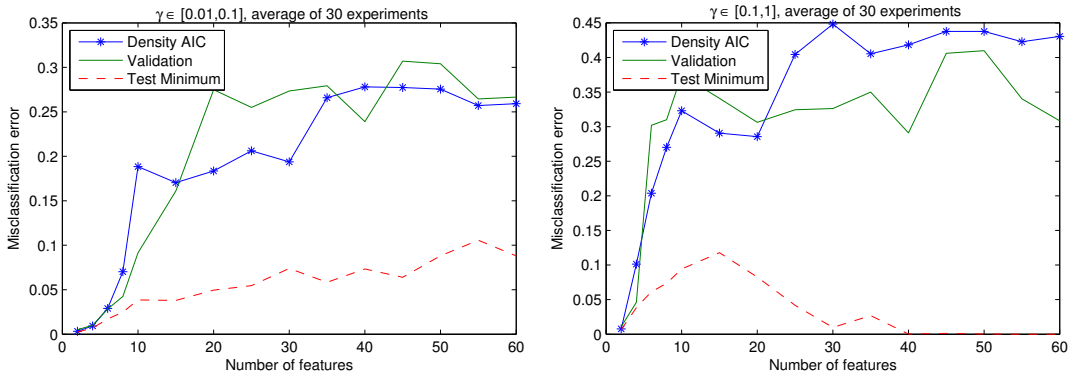

thyroid
Density AIC
Density BIC
Compression
Validation
Test Minimum
Average Error Rate
Dataset Size

5.6 Discussion of Results

We now discuss the plots from the previous section. We can see that all curves follow the "Test Minimum" shape almost everywhere, and therefore they appear to be reliable predictors for the best parameters. The single exception is the "Margin AIC" curve for the "banana" dataset, which shows adequate results only for dataset sizes ≥ 1200. In Table 2, we see that the "Density AIC" method achieved the best performance: it is consistently better than each of the other algorithms. Moreover, it appears to be the only one that is better than "Validation". As expected, the Density likelihood function seems to be a more accurate predictor, which however requires more computation time. Interestingly, the margin likelihood function, which is given for free, often demonstrates similar results. The comparison of AIC and BIC shows that AIC is better in the majority of cases, which might be caused by the relatively small size of the datasets from the UCI database.

While "Density AIC" clearly outperforms the other algorithms, there were two datasets where it did not converge to the "Validation" baseline ("splice" and "usdigits-0"). These datasets differ from the others in two aspects: they have the largest number of features among all considered datasets and a very low rate of unique values for features. We investigated the effect of the number of features in the following experiment. The results are presented in Fig. 5.3.

Figure 5.3: The plots of the test error for the predicted best values of the parameter γ on an artificial dataset with different number of features.



At first we constructed a set of points with the known best value of the kernel parameter γ. In order to do this we employed the following procedure. In each experiment (30 for each number of features) we randomly generated 800 points and their labels in the cube $[-2, 2]^d$ and the kernel parameter γ from a particular range ($[0.01, 0.1]$ or $[0.1, 1]$). After that we iteratively classified these points by the SVM classifier with this parameter γ and $C = 10$ and reassigned the labels until convergence. Finally, we generated the test set with $10k$ points that was perfectly separated by this classifier. Then we started the algorithms for searching the best parameters as they were described earlier. In the end the error rate of each algorithm was averaged for each number of features.

We can see that on the left plot of Fig. 5.3 (true $\gamma \in [0.01, 0.1]$) the Density AIC algorithm performs better than validation almost everywhere, but on the right plot (true $\gamma \in [0.1, 1]$) is starts to perform worse for $d \geq 25$. At this value of dimensionality the algorithm stops to recognize the right value of the parameter and predicts the same value as for the datasets from the left plot. We thus can claim that the higher dimensionality of the dataset with the fixed number of points, the lower the maximum value of γ that algorithm is still able to identify. The other experiment showed that the rate of unique values of features have no significant influence on the performance.

In order to solve this problem we tried to substitute Euclidean distance in the kernel feature space by Mahalanobis distance. In [88] the authors showed how to perform principal component analysis in the transformed feature space. Using their method, we can get new coordinates for all points with the dot product giving the same distance as the kernel function. Moreover, principal axes are centered and uncorrelated, so we only needed to divide the coordinates for each axis by its standard deviation. After that, according to the formula $d_M(x, y) = \sqrt{\sum_{j=1}^{d} (x_j - y_j)^2 / \gamma_j^2}$, the Euclidean distance for squeezed coordinates is equal to the Mahalanobis distance for the original coordinates. Since we considered only support vectors, we performed kernel PCA only for them. Having the dimensionality of the kernel feature space d, we left only d eigenvectors with the largest eigenvalues and considered coordination along only these vectors. However, our experiments have shown that this modification does not have any significant influence on the algorithm performance.

Another possible reason for the poor results is that was caused by the discrete nature of the data resulting in an excessive number of margin support vectors. When data have a continuous distribution in a d-dimensional feature space, it is impossible to get more than $d + 1$ margin support vectors, but it is possible in a case of a discrete distribution of features. This might cause incorrect identification of the dimensionality. We tried to solve this problem by a adding small amount of noise to the features. However, the value of noise required for the significant reduction of dimensionality was too high, leading to an inappropriate difference between the original and noisy data.

5.7 Adapation for neural networks

Information criteria have already been applied for tuning neural network structure ([67]). The proposed algorithm can also be employed for this purpose. The last layer weights w_K (2.3) can be retrained using the SVM for each output neuron independently $\hat{y}^i$ as one-vs-others classification. After that the AIC/BIC score can be computed as the average of the AIC/BIC scores for each neuron.

In order to determine the optimal structure, one starts to remove neurons with the lowest absolute weights, retrain the SVM classifiers, and compute the average AIC/BIC score. Given that the score curve is convex with a single minimum, it should be possible to speed up the process by using a quasi-gradient over the number of neurons. However, the described process seems to be too complex and computationally demanding to be used in practice.

5.8 Summary

We described two new approaches for using AIC and BIC to estimate the best values of parameters γ and C to use in an SVM with RBF kernel. Our approaches are based on two likelihood functions, derived from different perspectives. The first one is based on particular assumptions about the data distribution. It is less accurate, but computationally free. The second one analyzes the disposition of support vectors, directly computing the probability of the SVM

hyperplane to be a Bayes classifier. Our two approaches were embodied in four different algorithms: Margin-AIC, Margin-BIC, Density-AIC and Density-BIC. Among these four algorithms, the best performing was Density-AIC, which generally outperformed all others, including the more computationally expensive baseline methods "C3" compression code described in [65] and the simple "Validation" method.

Algorithm 5.1 Bayes likelihood computation

$-\log L(w|X,Y) \approx -\sum_{i=1}^{N_{SV}^{nm}} \log Pr\left[p(y_i|x_i) > 1/2\right]$

Implement $K(i,j) = RBF(x_i, x_j; \gamma)$

for $i \leftarrow 1$ **to** N_{SV}^{nm} **do**
    for $j \leftarrow 1$ **to** N **do**
        $r_{ij} = \sqrt{K(i,i) - 2K(i,j) + K(j,j)}$

for $i \leftarrow 1$ **to** N_{SV}^{nm} **do**
    $d_i = \sum_{j=1}^{N_{SV}} \lambda_j y_j K(i,j)$
$L \leftarrow 0$
for $i \leftarrow 1$ **to** N_{SV}^{nm} **do**
    if $(y_i d_i \geq 0)$ **continue**
    $r_i, y^s = sort(r_i)$
    $n_+, n_-, I_{min} \leftarrow 1$
    for $j \leftarrow 1$ **to** $min(N, N_{max})$ **do**
        if $(y_i d_j \geq 0)$ **continue**
        if $(y_i == y_j^s)$ **do**
            $n_+ \leftarrow n_+ + 1$
        else
            $n_- \leftarrow n_- + 1$
        $I_j = I_{1/2}(n_-, n_+)$
        if $(I_{min} > I_j)$ **do**
            $I_{min} \leftarrow I_j$
    $L \leftarrow L - \log I_{min}$
return L

Chapter 6

Deception detection

This chapter is based on the following publication: *Detection of deception in the Mafia party game. Sergey Demyanov, James Bailey, Ramamohanarao Kotagiri and Christopher Leckie. Proceedings of the 17th ACM International Conference on Multimodal Interaction, Seattle, Washington, USA, November 9-13, 2015*

6.1 Introduction

One of the goals of this thesis was to demonstrate the ability of machine learning algorithms to solve complicated classification problems. In this chapter we consider a very challenging problem of deception detection from visual cues. We introduce a new database of truthful and deceptive people, describe the algorithm of video processing, and propose a method of feature engineering that achieves a statistically significant level of accuracy: $AUC \approx 0.61$. This result is just a benchmark, that demonstrates the feasibility of the solution. We hope that the proposed algorithms of video processing and feature engineering will be improved in order to achieve higher accuracy.

Many organizations such as police, secret services, border security services and insurance companies depend on the recognition of deception and truthfulness of their clients. According to previous research ([9], [107], [125]), the average person detects liars with a probability that is statistically significant, but just slightly above a random chance. However, results of other experiments

have demonstrated that when people have a motivation to lie, their deception cues are present via four non-verbal channels: facial expressions, gestures and body language, verbal style and voice characteristics. Using these, trained people can achieve an accuracy of up to 73% ([24], [25]). Moreover, in [68] it is claimed that the analysis of facial expressions considered simultaneously with context could further increase this accuracy up to 90%, considerably higher than human performance. Since the face is the richest source of information, detection of facial cues of deception is one of the most important parts of such a deception detection system.

In this chapter we present a new database of truthful and deceptive people based on the videos of the Mafia party game (also known as Werewolf), and describe a methodology for feature extraction and classification of each person's role. In each game players are assigned to be either truthful or deceptive. At the same time there are no requirements on what the players have to say and how they should behave. A more detailed explanation of the game rules is given in Section 6.3. This new database contains 6001 labeled episodes from 270 participants with a total duration of 5 hours.

Our goal was to identify the players' roles based on their close-up face recordings. To create features we consider the first N minutes of each game, where N varies from 5 to 35 minutes. We extract features corresponding to the particular movements of facial muscles. Some of them are caused by experienced emotions, and therefore can be the signs of verity or the cues of deception. According to these features we build an automatic classifier of truthful and deceptive people.

First we give an overview of related work (Section 6.2), present the Mafia database (Section 6.3) and discuss the methodology of our research (Section 6.4). Next we describe the procedure of facial movement detection (Section 6.5), and explain the details of feature engineering (Section 6.6). Later we compare the obtained accuracy with the accuracy of a random classifier and demonstrate that the predictions are statistically significant (Section 6.7.1). Finally we analyze the most predictive features and show that they agree with the theory (Section 6.7.2). The conclusion (Section 6.8) finalizes the chapter. We believe our research can boost interest in the area of deception detection from facial cues.

6.2 Related work

The problem of deception detection has attracted considerable interest in recent years. Work in [107] established that the leakage of cues to deception is caused by the increased cognitive load experienced by liars, and therefore cannot be avoided. In [18] the authors analyzed 158 cues including facial expressions, linguistic features, physiological features and others. They discovered that people with higher cognitive loads are more likely to have less illustrators and body movements, more hesitations, longer pauses in speech, greater pupil dilation and more gaze aversion. Deceptive people can also experience a sudden increase of blood flow in the region of the eyes, which can be detected in thermal images [1]. Moreover, deception is often related with one of the 3 emotions: fear, guilt or delight [108]. The facial expressions of these emotions can also reveal these emotions.

In the experiments by Warren et al. [111], participants watched emotional and unemotional videos and were asked to lie about what they saw. The average classification accuracy for predicting lying was around 50%, however it was 64% for the group lying about emotional videos. This confirms the hypothesis that emotions can cause the non-verbal leakage of the deception cues. Their database is known as YorkDDT. Another database was collected by Frank et al. [30]. They recorded videos for an interrogation scenario with 100 participants of 2.5 minutes each. A multimodal database with 30 participants that includes video, thermal video, speech and physiological data have been collected in [79]. Mihalcea and Burzo [71] obtained 140 videos with truthful and deceptive people using Amazon Mechanical Turk. Zhang et al. [123] developed a system of expression classification based on facial key points and demonstrated its good performance.

A number of attempts have been made to develop an automatic system for deception detection. However, in most of them the features are not related to facial expressions. Several of them are based on linguistic features. For example, Fornaciari and Poesio used stylometric techniques to identify deception in the corpus of hearings collected in Italian courts [28]. Using only the linguistic features, Mihalcea et al. [72] could reach an accuracy $52 - 73\%$.

Thermal imaging approaches are also popular. Warmelink et al. [110] applied it to identify deception in airports. Their system demonstrated an accuracy higher than 60%. Rajoub et al. performed similar experiments with thermal videos on a set of 492 samples from 25 participants. While they were able to reach a very high level of accuracy of 87% for within-person predictions, the results of inter-person predictions (around 60%) were similar to other experiments. Another attempt to use thermal imaging has been performed by Jain et al. in [44]. They also obtained a classification accuracy for predicting lying of around 62%. In [1] the authors presented a system based on a multimodal approach, combining physiological features such as temperature, heart rate and pulse with linguistic features. The obtained accuracy was shown to be better than random, achieving 70% in some scenarios.

Some attention has also been paid to micro-expressions. The authors of [80] developed the system of micro-expression detection using LBP-TOP ([124]) features. They evaluated it on the corpus of video clips from [111] and their own database, and obtained quite promising results. Another database of 195 micro-expressions was collected in [119, 118]. The authors also provide the labels of the appearing action units with their timestamps. However, there were no attempts to use these databases and algorithms for deception detection.

6.3 The Mafia game database

We present a new database collected from the Mafia TV show, which contains 5 hours of videos from 270 participants. Unlike others, it was recorded in much more natural conditions than the experiments in other publications. Comparing with other deception detection databases in Table 6.1, it makes the Mafia database one of the largest. The source videos [1], the episode timestamps and player labels [2] are available online. We hope that it will be interesting for the research community and will boost progress in the area of deception detection.

The Mafia party game (also known as Werewolf) was invented in 1986 by the students of Moscow State University studying in the Department of Psy-

[1] `https://www.youtube.com/user/muzTV/search?query=mafia`
[2] `https://sites.google.com/site/mafiadatabase`

chology. This is how the game is described in Wikipedia: "[It is] modeling a conflict between an informed minority (the Mafia) and an uninformed majority (the innocents). At the start of the game each player is secretly assigned a role affiliated with one of these teams. The game has two alternating phases: 'night', during which the Mafia may covertly 'murder' an innocent, and 'day', in which surviving players debate the identities of the mafiosi and vote to eliminate a suspect. After elimination the player reveals his role. Play continues until all of the Mafia has been eliminated, or until the Mafia outnumbers the innocents". While the 'day' stage contains long discussions and voting, during the 'night' stage the 'Mafia' members silently show who they want to eliminate, and usually agree on a candidature within about 10-20 seconds.

The Mafia TV show series was shown on the Russian TV channel MuzTV in 2009 and 2010. The game participants were Russian celebrities and TV channel spectators. In total there were 30 series in this period [3]. The length of each of them is about 45 minutes. In each game there were 9 players. Two of them were from the Mafia team, the others were innocent. The role depends on the card the player gets at the beginning of the game. A black card corresponds to Mafia and a red card corresponds to innocents. Depending on the situation, each game had from 2 to 4 rounds of 'day'-'night' pairs.

Since during the 'day' stage the players from the Mafia team pretend to be innocent, we labeled them as deceptive. Others were labeled as truthful. Thus, our database contains 60 deceptive and 210 truthful players, 270 players in total. Note that the label of each player depends only on his role and is not changing within a game regardless of the current player's actions and words. Moreover, note that 'night' stage recordings are very short and do not contain discussions, so we excluded all 'night' appearances from our database.

6.4 Methodology

All facial expressions are caused by particular combinations of facial muscles. The list of these muscles is known from physiology. Based on this list, the

[3] 60 more series were shown later. Thus, the database can be increased by 3 times.

Table 6.1: The main parameters of deception detection video databases

database	Participants	Duration, min	Link
Mafia DB	270	300	
Amazon Turk DB	140	315	[71]
RU-FACS-1	100	250	[30]
Multimodal DB	30	75	[1]
YorkDDT	20	23	[111]

Swedish anatomist Carl-Herman Hjortsjö developed the Facial Action Coding System (FACS), that was later published by Paul Ekman et al. in 1978 and revised in 2002 [23]. This system defines 27 possible **Action Units** (AU) related with particular muscles (9 of them are in the upper part of the face and 17 in the lower part) and 6 basic emotions (fear, sadness, happiness, anger, disgust and surprise) that consist of different action units. The visual appearance of these action units can be found online [4].

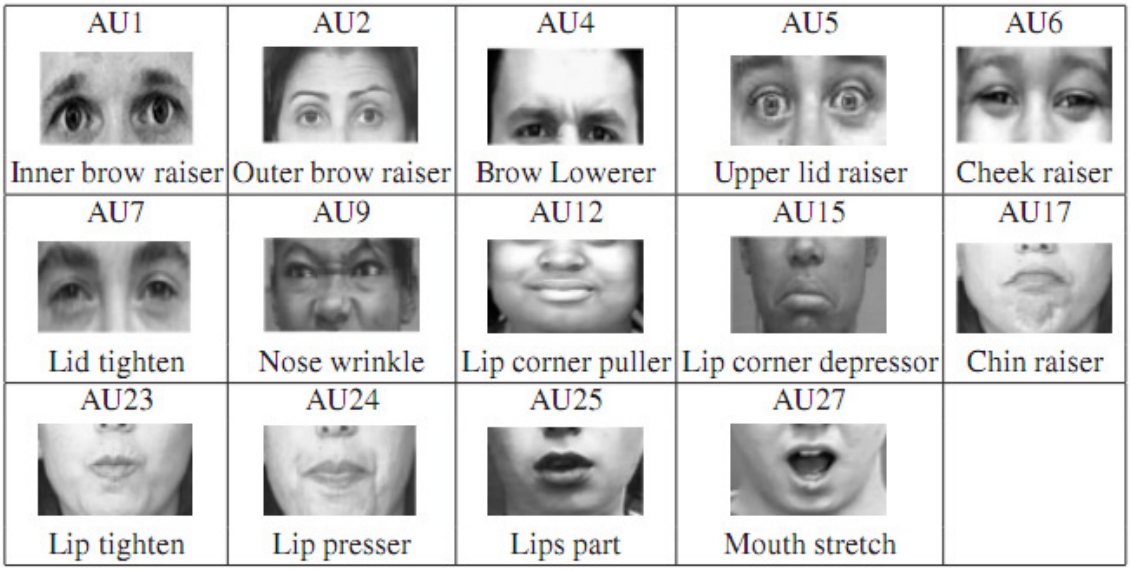


Figure 6.1: The examples of some Action Units extracted from the Cohn-Kanade database ([46]). The image is taken from [122].

Facial expressions can also be classified as *posed*, *spontaneous* or *concealed*.

[4] `http://www.cs.cmu.edu/~face/facs.htm`

Posed expressions appear deliberately in order to cause a certain impression. In contrast, spontaneous expressions appear unconsciously as a reaction on the ongoing events. Concealed expressions are also spontaneous, but their appearance is suppressed and therefore they have much smaller amplitudes.

In the book "Telling lies" [22], Paul Ekman provides a comprehensive analysis of the nature of deception. Chapter 5 describes the facial cues of deception. Ekman states that posed and spontaneous expressions are caused by different parts of the brain, and therefore have some subtle differences. One of them is that some facial muscles are involved only in spontaneous expressions and they cannot be readily inhibited. These muscles are called *reliable*. For example, only 10 percent of people can pull the corners of their lips down keeping their chin muscle fixed. The characteristic of being hard to suppress was called the inhibition hypothesis. This hypothesis was later confirmed in the experiments of [84].

Since facial expressions are supposed to be related with action units, we used this information to extract features. In order to do this we collected examples of action units and searched for their appearance in the Mafia database. The similarity scores of each player were used as features. We explain the full procedure in detail below. First we describe the algorithm of frame processing, and then provide the details of feature engineering.

6.5 Frame processing

Since the camera shows players from all angles and distances, players wear glasses and gesticulate in front of the face, as well as other difficulties, frame processing is not an easy task. For this purpose we developed a multi-stage automatic procedure which is briefly described in Algorithm 6.1.

6.5.1 Face detection

As the first element we employed the Viola-Jones algorithm [104] from the OpenCV library to detect faces. This library provides a detector that is already learned, so we could apply it straight away. In our experiments we considered

Algorithm 6.1 Frame processing

1. Face and eyes detection using OpenCV library
2. Initial in-plane rotation using eyes coordinates
3. Facial keypoint feature detection using Luxand FaceSDK
4. Width, height and angle normalization using keypoint features
5. Linear and non-linear image registration
6. Grid displacement computation for a sequence of frames.

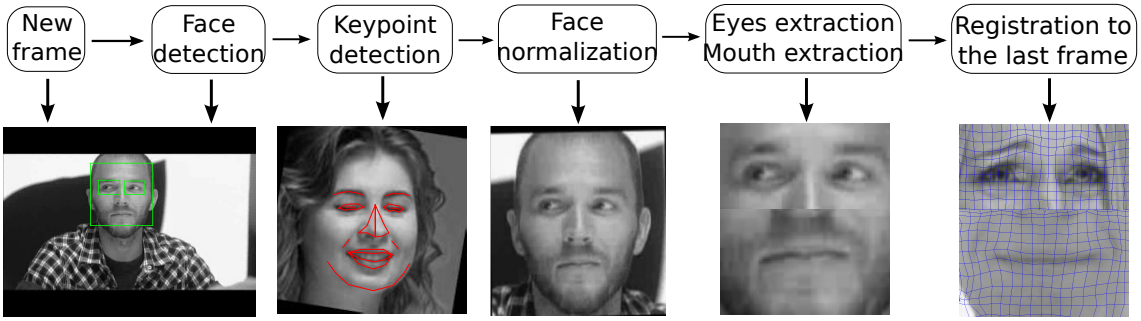


Figure 6.2: Visual illustration of the video processing pipeline.

only one face on each episode, so when we met more than one, we chose the one that is closest to the previously detected face. Once the face was detected, we applied the same algorithm for eye detection. The library provides different classifiers for the left and right eyes, so we approximately identified the regions for eyes and applied the classifiers in these regions. If no eyes were detected, we considered it as a false registration. In the case when only one eye was not detected on the current frame, but it was detected on the previous frame, we measured the displacement of the other eye and applied it to the current one. This situation happened quite often for people with glasses, and it allowed to treat such frames with the standard procedure. We used the position of detected eyes for initial in-plane rotation, so that the eyes become horizontally aligned.

6.5.2 Facial feature detection and normalization

After initial processing we applied the proprietary Luxand FaceSDK[5] software for facial feature detection. For a given face it returns the location of 66 facial points: 11 for each eye, 14 for mouth, etc. An example of the detected keypoints is shown on the Fig. 6.3. If the toolbox could not detect the features, we omitted such a frame. First we employed the obtained features to compute more precise coordinates of the left eye and right eyes (average of all left and right eye features, Fig. 6.3). These coordinates were used for additional in-plane rotation, so that the eyes are located on the horizontal line.

Second, we used these features to perform width and height normalization. To normalize width we computed the distance between eyes and scaled the image to make this distance equal to 80 px. Similarly we computed the visible nose height (the mean of the difference between y-coordinates of the left and right nose corners and its top), and scaled the image to make it equal to 50 px.

Figure 6.3: The examples of facial keypoint feature detection using Luxand FaceSDK. The eye features and nose top and corners are also visible. The eyes are horizontally aligned.

Third, we performed normalization with respect to other two types of rotation. While the in-plane rotation is a linear transformation and it can be easily suppressed using the eye coordinates, the other two out-of-plane types of rotation appear as a non-rigid transformation. In our problem we have only 2 regions of interest: eyes and mouth, so instead of registration of the whole face we cropped these two regions and treated them separately. These regions were

[5] `https://www.luxand.com/facesdk/`

considered as vertical cylinders with predefined radius values, so an up-down region rotation appears as a squeeze/stretch of an image in the vertical dimension, and therefore is a linear transformation. The same cylinder model makes it possible to handle a left-right rotation. We estimated the rotation angle using the eyes and nose coordinates obtained earlier, and computed how these cylinders would appear without rotation. In other words, we considered the visible face region as a projection of a cylinder on a rotated axis and computed its projection on the original axis. It leads to a non-linear stretching of the part that is closer to the observer and otherwise.

6.5.3 Image registration

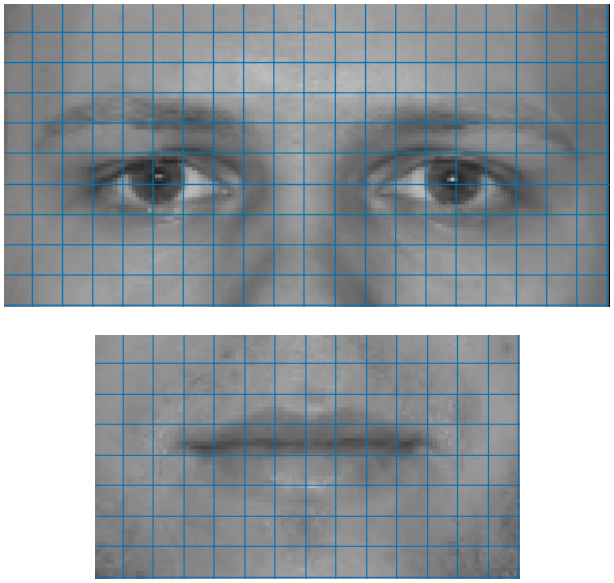


Figure 6.4: An example of normalized eyes and mouth with the 8×8 uniform grids

After face normalization we perform image registration. It allows us to obtain a description of non-linear movements within a face, that are related to action units. This procedure is based on the method of nonrigid registration using free form deformations, described in [86], and used in [50].

The problem is the following: for two similar images we want to find a non-

linear transformation that maps one image into the other one. As in [50], we also use a sum of squared distances between pixels as a similarity measure. We model the transformation by the displacement of nodes of the uniform grid in Fig. 6.4 with the size of a cell 8×8 px. For the displacements p_k of the nodes of the uniform grid x_k, the value for the pixel x is taken from the pixel $T(x)$ of the original image:

$$T(x) = x + \sum_{x_k \in N_x} p_k \beta \left(\frac{x - x_k}{\sigma} \right),$$

Here N_x is the set if 16 nodes around the point x, p_k's are the displacements of these grid nodes, β is the cubic multidimensional b-spline polynomial function and σ is the regular grid spacing (8). To avoid problems with the corner pixels, the regular grid has 2 rows and columns outside the image from each size. Thus, the parameters of the model are the coordinates of the extended uniform grid nodes ($16 \times 26 \times 2 = 832$ variables for the eye region and $14 \times 20 \times \times 2 = 560$ variables for the mouth region). The minimization problem is solved using the L-BFGS algorithm.

Before solving non-rigid registration problem, we first equalize the intensity histograms in order to avoid variations in brightness. Second, we perform affine registration, which also minimizes the SSD measure. It gives us the 3×3 affine transformation matrix, which aligns frames linearly. The non-linear transformation is performed in two steps with different grid spacing. First the SSD measure is minimized only for a half of nodes (i.e., using double spacing), and after it the solution is updated using all nodes.

Because non-rigid registration works well only when the difference between frames is small enough, we applied it only to nearby frames. To obtain the transformation function between two frames on an arbitrary distance, we sequentially applied the grids for frames between them. In other words, if we know all functions T_{i-1}^{i} for $i = 1 \ldots N$, then the function T_0^N is just their composition, i.e.

$$T_0^N(x) = T_{N-1}^N(\ldots T_1^2(T_0^1(x)) \ldots).$$

Here T_{i-1}^{i} is the function that transforms the points from the frame $i-1$ to the frame i. However, this process is not precise and good results can be obtained

only for a sequence of not more than 30 frames. Note that the functions $T(x)$ are defined only for pixels in the image, while the output $T(x)$ may be outside the image. In this case we do not have 16 nodes around this output (that is an input for the next function) and we have to approximate the displacements for these non-existing nodes. This leads to inaccurate results of the points near the border.

Fig. 6.5 provides an example of the cumulative grid displacement for 17 frames, containing the appearance of Action Unit 1. We can see that for this number of frames the obtained representation of facial movements is quite accurate.

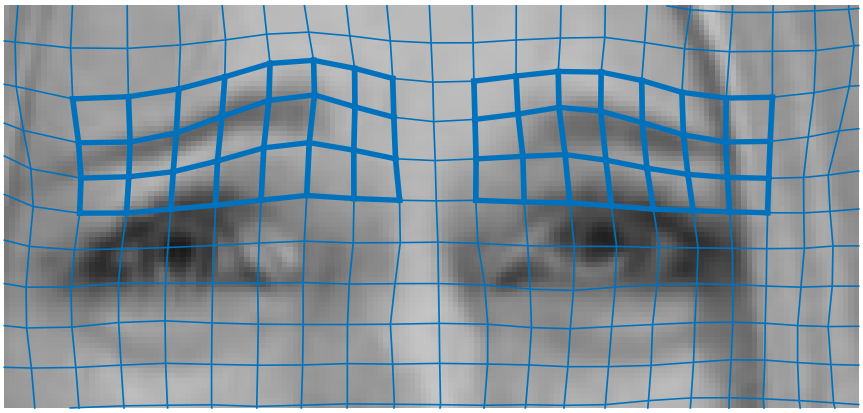


Figure 6.5: An example of grid modification after 10 frames from the game 17 with the player 4. The example represents the onset of Action Unit 1. Bold nodes are used for AU1 similarity search.

6.6 Feature engineering

Algorithm 6.1 allows us to compute the displacement of grid nodes in a sequence of consecutive frames. These node displacements give a compact representation of the movements between the first and the last frames. Here we describe the procedure of feature extraction using this information. Its scheme is given in Algorithm 6.2.

Algorithm 6.2 Feature extraction

1. Extract all episodes from the Mafia videos using Algorithm 6.1
2. Choose a collection of AU examples from the MMI database
3. Compute the displacement grids for the onsets and offsets of chosen examples using Algorithm 6.1
4. Find the subsequences of the episodes that are most similar to the AU examples
5. Compute their similarity scores
6. Aggregate the similarity scores over all episodes of each player (choose the minimum)

6.6.1 Episode extraction

We processed all source Mafia videos to extract the sequences of neighboring frames that are registered to each other using Algorithm 6.1. We refer to them as *episodes*. However, within a single episode some frames might be omitted for different reasons. For example, it might happen because the face is occluded by a hand and therefore it is not detectable. Their number should be small enough to have the registered frames to be close to each other and large enough to keep the episodes continuous, for example, in the situations of people with glasses. In our case the maximum allowed number of consecutive omitted frames in a single episode was 10. It the number was larger, we ended the episode and started a new one. Given that each pair of frames required about 3 seconds for registration, it was the most time consuming part of the total pipeline.

We extracted episodes in the interval from 7 to 42 minutes, when all the game actions were taking place. In total we obtained 6733 episodes in 30 games with an average of 224 episodes per game, 29 per player. The minimum and maximum number of episodes in one game were 175 and 284. The average length of each episode was 77 frames, i.e., about 3 seconds. Each episode was manually viewed and labeled according to the player appearing on it. If the episode was corrupted, contained non-players or players after the game, it

was labeled as 0 and was omitted in further calculations. The total number of episodes after elimination is 6001.

6.6.2 MMI database

The MMI database ([78]) contains a wide range of videos and images representing different emotions. Some of the videos are also labeled according to the Action Units they contain. Additionally they contain the information about AU stage (onset (or start), peak, offset (or finish), neutral) for each frame. We employed this information for feature selection.

Theory suggests ([22, 84], that the action units caused by reliable muscles (like AU1) correspond to felt emotions. Depending on the emotion it can be a sign of either truthfulness or deception. Therefore, it was plausible to find the appearance of such action units. For this purpose we employed the same image registration procedure for MMI examples as for the Mafia database. We selected a maximum of 3 examples of non-empty onsets and offsets for each of the 14 most common action units (namely 1, 2, 4, 5, 6, 7, 9, 12, 15, 16, 17, 20, 23 and 24), 79 in total. The indices of video clips with AU examples from the MMI database including the duration of onset and offset stages are presented in the Table 6.2. The choice of 3 examples provides a balance between the variety of AU representation and the generalization ability of the trained model. AU templates are similar, and the corresponding features are highly correlated. If the number of features was too large, this could result in overfitting behavior by standard classifiers such as the employed logistic regression. The examples were chosen randomly (in fact, in lexicographical order of their numbers) in order to avoid inflation of reported performance due to over tuning of feature selection.

For each example we performed image registration of the frames with the 'onset' and 'offset' labels. Then we used the computed transformation functions to compute the new coordinates of the uniform grid nodes on the last frame of the resulting sequences. For each of these sequences we also computed the cumulative linear transformation by multiplying the transformation matrices for each frame in the sequence. The new coordinates were multiplied on the in-

verse cumulative transformation matrix in order to suppress head movements. Since different action units appear in different areas of the eyes and mouth, we used only the subset of nodes located in these areas. The list of these nodes for each AU is also provided on the database website. The displacements of these nodes uniquely identify the appeared action unit.

There might be some concerns about the validity of usage of posed AU examples from the MMI database in order to find the spontaneous appearance of AU in the Mafia database. However, our algorithm takes the difference between posed and spontaneous expressions into account. More specifically, that most of the difference lies in the temporal dimension, which is effectively handled in our algorithm by considering all possible pairs of first and last frames to detect action units. Second, our algorithm detects the presence of AUs, rather than their type. AU type is another discriminative factor, which might further boost accuracy, but is more difficult to correctly detect. The employed algorithm is more preferable than approaches like CERT ([63]), where the AU appearance scores are based on each frame independent of others. On the other hand, our displacement grids are based on changes in time, thus containing richer information, which is exploitable in our framework.

6.6.3 Feature extraction

The next step was to find the appearance of action units in the extracted episodes. Since they might have appeared at any moment, we considered all subsequences of consecutive frames of total length not more than 30. For each of them we computed the displacement of the subset of uniform grid nodes specific for this AU, the same way as we did for MMI examples. Then we computed the Euclidean distance between the displacement vectors of each episode subsequence and each onset and offset of examples of the action units from the MMI database obtained in the previous step.

We assume that within a single episode a particular action unit can appear only once. Given that, we computed the minimum distance among all subsequences and recorded it as a feature for this episode. Thus, for each episode we obtained 79 features (40 onset and 39 offset features). For convenience we

Table 6.2: The indices of Action Units from the MMI database, which are used as examples. The numbers of onset and offset frames are provided in the brackets

AU	Examples (onset and offset duration)		
1	1931 (3, 0)	24 (18, 18)	582 (4, 6)
2	144 (3, 9)	145 (10, 13)	1649 (6, 8)
4	1047 (4, 7)	1384 (5, 3)	1823 (6, 13)
5	1 (5, 9)	1275 (6, 11)	144 (3, 9)
6	1074 (5, 5)	1088 (7, 8)	123 (11, 16)
7	1316 (0, 2)	1874 (3, 7)	1973 (5, 6)
9	1384 (5, 5)	1964 (11, 10)	199 (3, 5)
12	123 (17, 37)	124 (28, 39)	125 (12, 19)
15	1077 (7, 3)	1152 (7, 7)	1153 (5, 5)
16	134 (5, 6)	135 (7, 17)	14 (11, 18)
17	1152 (8, 6)	1153 (4, 3)	12 (10, 14)
20	1088 (7, 8)	1812 (13, 4)	1813 (10, 7)
23	382 (8, 4)	611 (12, 12)	
24	1874 (11, 8)	1931 (3, 0)	1973 (4, 6)

recorded them with a negative sign. We will refer to them as the similarity scores.

The next goal was to create features for each player in each game. As before, we used maximum to aggregate the scores over each player episodes. We thus selected the maximum similarity scores over all episodes for a particular player and consider them as features for this player.

While a higher similarity score corresponds to a higher probability of AU appearance, the similarity scores are still quite noisy. In fact, after the visual examination we determined that only the first quartile of their values correspond to the episodes that are similar to the related action unit. Other 3 quartiles seemed to contain random episodes, regardless of the score value. To incorporate this knowledge in the dataset we subtracted the 78% - percentiles (7/9, the percentage of truthful players) and set all the negative values to 0. Therefore, we produced a sparse dataset with no difference in the feature values between non-top values.

6.7 Experimental results

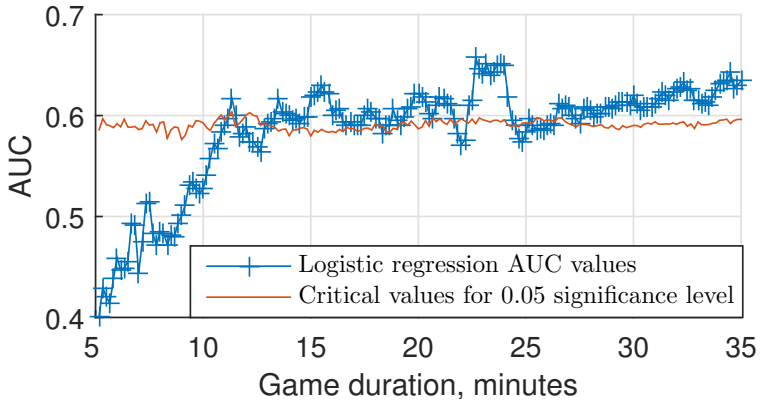
During the game some players are eliminated and do not appear in the later stages. This causes an imbalance in the total observation time. Moreover, the eliminated players are more likely to be truthful, because only truthful players are eliminated during the 'night' stage. This might also introduce a bias in the features. In order to validate the obtained results we created a number of datasets, corresponding to the different game duration. We split all 35 minutes on 10 second intervals and considered game durations in the range from 5 to 35 minutes, totally 181 intervals. For each of them we selected only those episodes that completely fall into this interval. Thus, each set of these episodes gives an independent dataset.

6.7.1 Classification

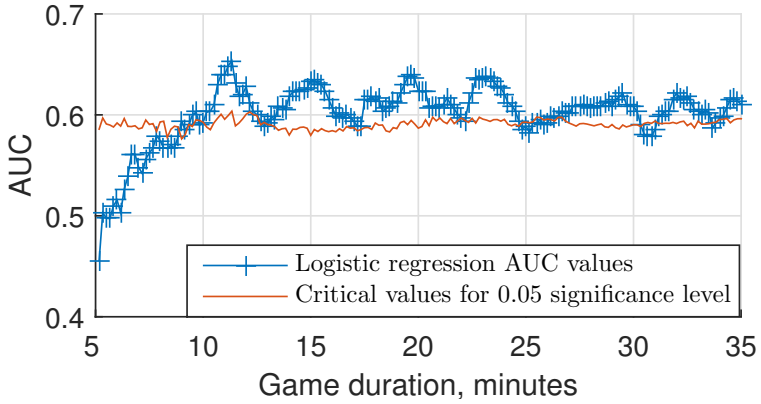
Since the dataset is highly unbalanced, we used the area under the ROC curve (AUC) as a performance metric instead of classification accuracy. We employed a simple logistic regression classifier ([42]). Before classification we eliminated the players without observed episodes. There are 22 of them for the 5 minute game duration, and only one (player 3 in the game 16) for the 35 minutes. We performed 30-fold cross-validation, every time leaving out the players from a single game. Thus we kept the same proportion of the players in the training and test sets as in the full dataset.

Fig. 6.6 presents the plots of overall AUC depending on the game duration. It demonstrates a clear increase of performance in the range from 5 to 12 minutes, that corresponds to the increase of the number of considered episodes. This is a natural behavior: once we take more information into account, the predictions become more and more accurate. After that the AUC remains on the same level around 0.61. The maximum duration of 35 minutes gives the $AUC = 0.6391$. It corresponds to the accuracy 70.26%, while the random predictions give the average accuracy 65.35%. Choosing a classification threshold such that the number of predicted positive examples is equal to the one in the training set,we get the same corresponding precision and recall 0.8082.

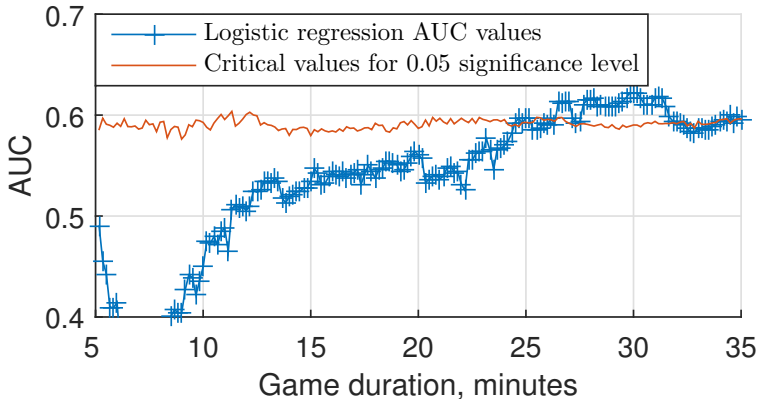
Figure 6.6: Plots show the AUC (area under ROC curve) values as a function of game duration for eyes, mouth and all features. The orange line represents the critical values for the 0.05 significance level.



(a) All features



(b) Eyes features



(c) Mouth features

Therefore F1-score is also 0.8082. From Fig. 6.6 (b,c) we can see that eyes features independently perform better than mouth features, which reach the significance level only after 25 minutes. The reason might be a larger variety of mouth movements, caused by speech. Since it acts as noise, more time is required to collect meaningful statistics. We also evaluated the performance of onset and offset features independently. As expected, the results are worse than when they are combined together. However, onset features appear to be more predictive. Similar results were obtained using a linear SVM classifier with the regularization parameter $C = 1000$.

In order to assess the statistical significance of the results, we computed the critical AUC values. We tested the hypothesis that such AUC values could be obtained for a random distribution of labels. For each dataset, corresponding to a particular game duration, we performed the classification with the randomly permuted labels, 400 permutations. Permutations allowed to preserve the ratio of positive and negative instances. The critical values for the 0.05 significance level are presented by the orange curve in Fig. 6.6. As we can see from it, the obtained AUC is higher than the critical curve for almost every dataset based on more than 13 minutes. As it follows from the definition of critical values, the probability of this to be random is less than 0.05. It confirms that computed features are indeed the label predictors, containing relevant information.

We also computed the mean of AUC for all time intervals $5 - 35$ (0.5849) and time intervals $15 - 35$ (0.6106). Using the same randomization, we computed their p-values: 0.0104 and 0.0026 accordingly. Both of them are much lower than 0.05, indicating a result due to random chance is unlikely.

It is interesting to compare the obtained accuracy with a human baseline. We can compute a rough estimation the following way. Recall that a game consists of repeating 'day'-'night' phases. While in the 'day' phase players try to eliminate a deceptive player, in the 'night' phase they always lose one truthful player. For a predefined number of players (7 vs 2) there is a limited number of possible outcomes (10). Assuming a random choice of eliminated players in the 'day' phase, we can compute the probability of each of these outcomes. The Mafia team wins in 4 of them, with the total probability equal to $221/315 \approx 70.16\%$. In fact, the Mafia team won in 21 among 30 games, i.e., exactly in 70% of cases.

It confirms that people detect deception very close to random, even when they get information from all channels including video, audio and context. However, the same result was established earlier in psychological research [9].

6.7.2 Feature analysis

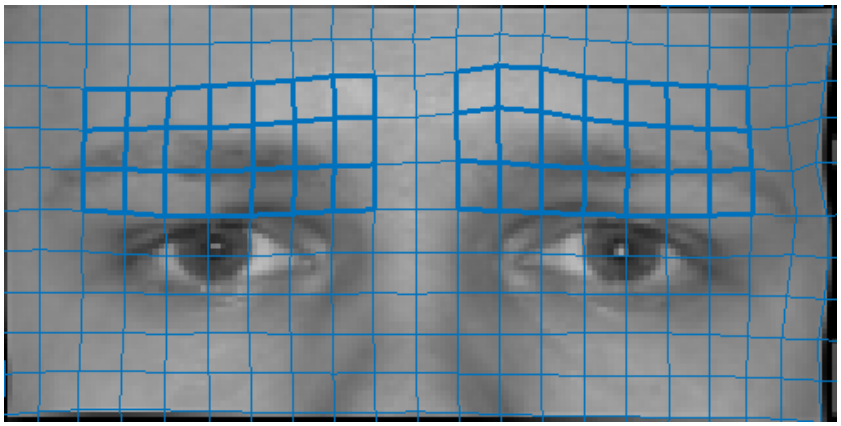
The logistic regression classifier also provides p-values for the obtained coefficients. We used these coefficients to identify the most statistically significant features. The top 3 of them are presented on Fig. 6.7. Note that all of them are caused by the reliable muscles [70], and therefore are difficult to be simulated. We also computed the corresponding feature coefficients. While their amplitude depends on the similarity scores and does not provide any information, their signs show how the feature influences the final prediction. Positive coefficients increase the probability of truthful people, while negative ones decrease it. Here we use the average of coefficients over all 30 folds and game durations from 15 to 35 minutes, when the datasets are based on a sufficient number of episodes.

The first of the top 3 features is an example of the onset of Action unit 1 (inner brow raiser). It has a positive coefficient, with the p-value 0.030. The second feature represents the offset of the AU20 (lip stretcher). It has a negative coefficient with the p-value 0.037. The third feature is connected with the AU16 (lower lip depressor). It also has a positive coefficient. Its p-value is 0.039.

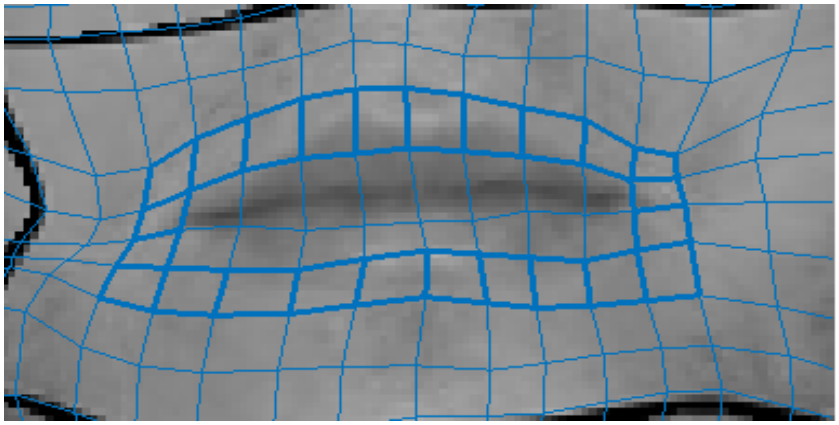
We can provide a possible explanation of these results. Theory says that deceptive people tend to experience fear, guilt or delight [108]. Opposite to that, sadness has been shown to be a sign of verity [91]. It is known [23] that AU1 and AU16 might be caused by sadness, while AU20 can appear as a result of fear. Therefore, their coefficient signs do not contradict the theory.

We also tested the performance of these 3 most significant features independently. None of them achieved statistically significant classification accuracy, which confirms that only a combination of cues can give a reliable result. This was earlier stated in [108].

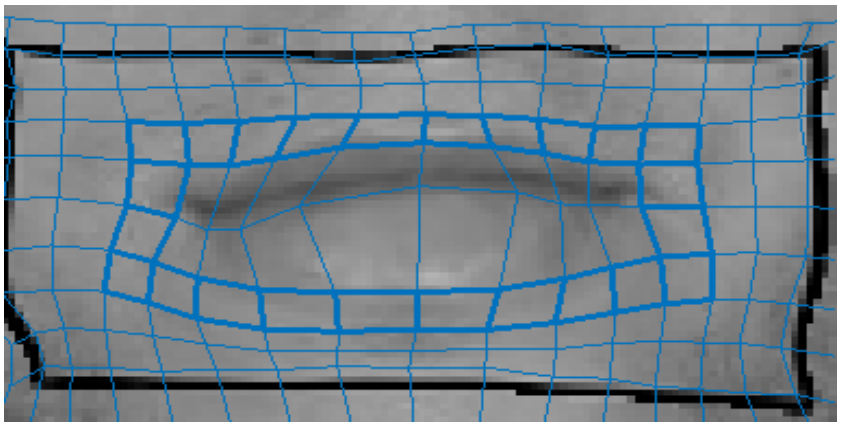
Figure 6.7: MMI indices of examples of action units related with 3 the most significant features.



(a) 582(AU1), onset, truthful, p-value = 0.030



(b) 1088(AU20), offset, deceptive, p-value = 0.037



(c) 135(AU16), onset, truthful, p-value = 0.039

6.8 Conclusion

In this chapter we presented an automatic system of deception detection based on the features extracted from movements of eyebrows, eyes and mouth on videos. We demonstrated that these features contain sufficient information to achieve a classification accuracy significantly higher than random predictions. The list of the most predictive features can be explained according to the results previously established in the psychological literature. We also introduced the new Mafia database of truthful and deceptive people, which was recorded in more natural conditions compared to previous studies. This database contains in total 5 hours of video, which make it one of the largest available. We hope that it will become a benchmark for assessing algorithms for deception detection. The demonstrated performance is not good enough to be used in practice, but it establishes a lower bound for the future.

This research is in its beginning. Future research might improve all parts of the presented system. For example, one might explore developing more accurate methods of action unit detection, extraction of more meaningful features or usage of other modalities such as voice tone and language features. The multimodal approach that combines all types of features seems to be the most promising. Another significant drawback of the algorithm is its speed. The process of image registration takes about 3 seconds per frame, which does not allow it to be used for real-time predictions. This procedure might be accelerated by using less expensive features or more powerful hardware like a GPU. It would be a big step forward to make the algorithm work in real-time without loss of accuracy. Moreover, one can increase the database by processing all other 60 videos and labeling extracted episodes. That would make the database the largest one. We believe that this research will stimulate interest in the challenging problem of automatic deception detection.

Chapter 7

Conclusion

7.1 Summary of contributions

This thesis we focused on regularization methods for neural networks. The topic has been gaining increasing importance as it becomes possible to train very deep neural networks with millions of parameters, which can easily overfit. In tasks with a limited amount of data they might be crucial to achieve good performance. In other tasks they might significantly reduce the required training time. Due to the increasing popularity of neural networks, a number of new regularization techniques have been proposed in recent years. However, there has been no overview that would describe them together so far. In this thesis we have aimed to provide an overview of the well-known methods as well as the recent ones, and proposed several algorithms and tasks related to this topic.

First we provided the background for the neural networks. We started from the simplest model of a single perceptron, and its learning rule, and finished with the multiclass multilayer network with the softmax output layer, and the matrix notation for the backpropagation algorithm with the negative cross-entropy loss function, which is used in classification tasks. Next, we described the problem of the bias-variance tradeoff. We gave a definition of model complexity as the VC-dimension, and explain the phenomena of underfitting and overfitting. We introduced the definition of regularization methods as a way to decrease the model variance that preserve the bias, and show that many of them have a

Bayesian interpretation.

Second we proposed the **Invariant Backpropagation algorithm** (Chapter 4). It is an extension of standard Backpropagation, which **minimizes the main loss function L_{train} together with the additional regularization term L_{inv}**. This term is computed at the end of the backward pass, using the gradients of the main loss function with respect to the input vector $\partial L_{train}/\partial x$. We showed how to efficiently compute the gradients of L_{inv} with respect to the parameters $\partial L_{inv}/\partial w$. This computation requires an additional forward pass, which can be implemented using the initial forward pass functions with minor modifications. The additional weight gradients $\partial L_{inv}/\partial w$ are computed the same way as the main gradients $\partial L_{train}/\partial w$. As a result, in practice the algorithm requires only about 50% more computation time than standard backpropagation, improving the accuracy up to 30% on small datasets without data augmentation.

We also compared IBP with the two most similar approaches: Tangent BP and Adversarial Training. We demonstrated that **IBP can be considered as a computationally efficient version of Tangent BP** with another loss function, which makes a classifier robust to all variations, while Tangent BP implies robustness only to the particular directions given by tangent vectors, which have to be computed in advance. The comparison with Adversarial Training shows that they both minimize the main loss function together with its first gradient with respect to the input, but the Adversarial Training algorithm also minimizes all higher order derivatives with the coefficients, given by the Taylor expansion. The experimental results show that it might be an advantage on some datasets, and a disadvantage on others. At the same time, we showed that the algorithm of **Adversarial Training** as it is described in [33] **is inefficient and can be sped up on** $\approx 20\%$ by avoiding the computation of gradients for the main loss function.

Third, we analyzed a method of model selection based on AIC/BIC criteria for the SVM classifier with the RBF kernel. These criteria contain two values: the likelihood of the model for the given data, and the effective number of free parameters. We proposed two ways to compute **the likelihood function**. The first way is to use **the SVM objective**, which is already computed when the SVM solution is found. We show that the SVM objective can be considered as

the likelihood of a model with the particular data distribution. In more detail, the probability of the support vectors should exponentially decrease as a function of their distance from the hyperplane. The contribution of non-support vectors to the likelihood should be zero. The second way assumes the direct estimation of the probability that the SVM hyperplane is a Bayesian classifier. In other words, the density of the positive/negative class should be higher in every point of the space classified as positive/negative. We estimate this probability at the location of every support vector using the number of positive and negative examples in their neighborhood, and show that it can be computed using **the regularized incomplete Beta function**, which can be efficiently calculated. Another component is **the effective dimensionality of the SVM classifier**. We show that for the data sampled from the continuous p.d.f. it **is equal to the number of margin support vectors**, which can be easily found by the corresponding values of the coefficient $0 < \lambda < C$. In the end we evaluate four variants of the algorithm given by the combinations of the AIC/BIC criteria with the Margin/Density likelihood functions, and demonstrate that Density AIC is able to find the optimal parameter of the RBF kernel better than the similar methods and cross-validation. At the same time, Margin AIC, which is computationally free, shows comparable performance, and therefore can also be used for model selection instead of cross-validation.

Forth, we investigated a challenging problem of deception detection from visual cues. For this purpose we collected a dataset of truthful and deceptive people, based on the Mafia party game TV show. The rules of the game assume that people are split into two teams, and people from one team have to pretend that they belong to another one. In other words, some people have to be deceptive, while others have to prove that they are not. The TV show allows us to know the ground truth in the end, so it becomes a well-posed problem of supervised learning. We process 30 videos of 45 minutes length, and extract 6001 episodes of facial close-ups, which are later labeled according to the person appearing on it. The process of video processing can be represented as a pipeline with several steps. First, we detect the face using the well-known algorithm of Viola-Jones [104], implemented in the OpenCV library. The same algorithm is employed to detect the eyes. Second, we find the facial keypoints using

the Luxand SDK toolbox. It outputs the coordinates of 66 keypoins, located on the eyes, nose and mouth. These coordinates are further used for the third step - face normalization, which adjusts the width, height and rotation angles. The last step is non-rigid image registration. It approximates the changes between consecutive frames by the movements of the uniform grid. Accumulated through several frames, they can represent typical facial movements caused by particular muscles. These movements are called Action Units. The dataset of extracted episodes is our first contribution.

Next, we described the method of feature engineering. For this purpose we employ another dataset (MMI), which contains examples of typical action units. We process these examples the same way as the episodes of the Mafia database, and obtain the displacement grids, which represent the corresponding AUs. These grids are then used as filters in order to find the most similar episode subsets. The similarity scores are later used as features for classification. We show, that this allows us to obtain an accuracy that is slightly, but statistically significantly better than random predictions. This result is our second contribution. We hope that the presented dataset and the baseline for it will boost further research in this area.

7.2 Notes on Future Work

Our results open several directions for future work. In this final section of the thesis we first point out the limitations of the presented work, and then highlight some potential avenues of future research.

7.2.1 Invariant Backpropagation algorithm

While the presented Invariant Backpropagation algorithm is an efficient method to achieve model robustness, it introduces one more regularization parameter that needs to be tuned in order to achieve an improvement in accuracy. One can try to develop a method of automatic tuning of this parameter based on the current performance of the classifier. In fact, the problem of hyperparameter tuning is more general. While a number of methods have been proposed (for

example, [92], [4], [66]), hand-tuning remains the most popular approach so far. At the same time, the network structure might be crucial for its performance. For example, the best result in the ImageNet 2014 competition was achieved by using very deep networks with very small filters 3×3 ([90]). Therefore, the problem of tuning the particular hyperparameters, including the number of layers and neurons, as well as the problem of tuning hyperparameters all together, remain open.

Another open research question is the properties of the datasets that make Invariant BP preferable to Adversarial learning and vice versa. Our experiments have shown that none of them is preferable in all situations, so it is important to understand when each of them should be used.

7.2.2 Tuning of SVM hyperparameters

The proposed algorithm of SVM parameter optimization is based on the properties of the trained classifier. It allows us to estimate the performance for current hyperparameters, but does not allow us to obtain the gradient of the AIC/BIC criteria with respect to these hyperparameters. Therefore, the grid search over a set of values is still required. One can try to adapt the proposed algorithm in such a way that gradient descent over the space of hyperparameters becomes possible.

It also seems interesting to extend the proposed algorithm on other classifiers, such as neural networks, without retraining the last layer parameters using the SVM objective. While the method of density likelihood computation is general, the margin likelihood and the number of parameters is specific for the SVM classifier. It seems reasonable to keep using the classifier objective function as likelihood, because it is computationally free, and it can often be interpreted as a likelihood for a particular data distribution, as we have shown for SVM. However, the effective number of parameters in the case of neural networks is not obvious.

7.2.3 Deception detection

In Chapter 6 we proposed the algorithm of feature engineering, which allows us to achieve an accuracy that is statistically significantly better than random. This result opens the door for further improvement. This can be done in many ways. Our algorithm obtains the similarity scores for the examples of different action units. First, one can employ other algorithms of action unit detection, which are not based on the direct comparison with the MMI examples. Second, the algorithm of feature extraction does not have to rely on action units at all. One can use unsupervised processing of the extracted displacement grids in order to obtain more meaningful representation. This can be done using a simple approach as K-means clustering, which has been proved to be efficient in the image ([14]) and video ([55]) classification tasks. One can also try to use a recurrent neural network (Section 2.2.11), which uses the next frame as a label for the previous sequence of frames. The aggregates of the obtained hidden states over time can also be used as features.

Third, the deep learning models can be employed for learning directly from source videos. This seems to be too difficult for supervised learning due to the extremely complicated classification function and the limited size of the training set (270 participants). However, it might be possible to extract the most typical spatio-temporal patterns in an unsupervised manner ([57], [97]), and use the statistics of their appearance for supervised learning. If the extracted patterns appear to be similar with examples of action units, it will be an excellent proof of their predictive power. Alternatively, convolutional neural networks can be trained for action unit detection in spatio-temporal domains on some large dataset. This approach demonstrated good results in the video classification domain ([48]). The output of such classifier might be later used to predict deception. However, a labeled dataset of action units with millions of examples might not be available for a long time.

A parallel way to improve accuracy is to use voice as another source of information.

Bibliography

[1] Mohamed Abouelenien, Veronica Pérez-Rosas, Rada Mihalcea, and Mihai Burzo. Deception detection using a multimodal approach. In *Proceedings of the 16th International Conference on Multimodal Interaction*, pages 58–65. ACM, 2014.

[2] Peter L Bartlett and Shahar Mendelson. Rademacher and Gaussian complexities: Risk bounds and structural results. *The Journal of Machine Learning Research*, 3:463–482, 2003.

[3] Yoshua Bengio, Ian J. Goodfellow, and Aaron Courville. Deep Learning. Book in preparation for MIT Press, 2015.

[4] James Bergstra, Daniel Yamins, and David Cox. Making a science of model search: Hyperparameter optimization in hundreds of dimensions for vision architectures. In *Proceedings of The 30th International Conference on Machine Learning*, pages 115–123, 2013.

[5] Dimitri P Bertsekas. *Constrained optimization and Lagrange multiplier methods*. Academic press, 2014.

[6] HS Bhat and N Kumar. On the derivation of the Bayesian Information Criterion. *School of Natural Sciences, University of California*, 2010.

[7] Christopher M Bishop. Training with noise is equivalent to Tikhonov regularization. *Neural computation*, 7(1):108–116, 1995.

[8] Christopher M Bishop. *Pattern recognition and machine learning*. Springer, 2006.

[9] Charles F Bond and Bella M DePaulo. Accuracy of deception judgments. *Personality and social psychology Review*, 10(3):214–234, 2006.

[10] Arthur E Bryson. A gradient method for optimizing multi-stage allocation processes. In *Proc. Harvard Univ. Symposium on digital computers and their applications*, 1961.

[11] Kenneth P Burnham and David R Anderson. *Model selection and multimodel inference: a practical information-theoretic approach*. Springer Science & Business Media, 2002.

[12] Chih-Chung Chang and Chih-Jen Lin. LIBSVM: A library for support vector machines. *ACM Transactions on Intelligent Systems and Technology (TIST)*, 2(3):27, 2011.

[13] Olivier Chapelle, Vladimir Vapnik, Olivier Bousquet, and Sayan Mukherjee. Choosing multiple parameters for support vector machines. *Machine learning*, 46(1-3):131–159, 2002.

[14] Adam Coates, Andrew Y Ng, and Honglak Lee. An analysis of single-layer networks in unsupervised feature learning. In *International conference on artificial intelligence and statistics*, pages 215–223, 2011.

[15] George Cybenko. Approximation by superpositions of a sigmoidal function. *Mathematics of control, signals and systems*, 2(4):303–314, 1989.

[16] Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. Imagenet: A large-scale hierarchical image database. In *Computer Vision and Pattern Recognition, 2009. CVPR 2009. IEEE Conference on*, pages 248–255. IEEE, 2009.

[17] Li Deng, Jinyu Li, Jui-Ting Huang, Kaisheng Yao, Dong Yu, Frank Seide, Michael Seltzer, Geoff Zweig, Xiaodong He, Jason Williams, et al. Recent advances in deep learning for speech research at Microsoft. In *Acoustics, Speech and Signal Processing (ICASSP), 2013 IEEE International Conference on*, pages 8604–8608. IEEE, 2013.

[18] Bella M DePaulo, James J Lindsay, Brian E Malone, Laura Muhlenbruck, Kelly Charlton, and Harris Cooper. Cues to deception. *Psychological bulletin*, 129(1):74, 2003.

[19] AR DiDonato and MP Jarnagin. The efficient calculation of the incomplete beta-function ratio for half-integer values of the parameters. *Mathematics of Computation*, 21(100):652–662, 1967.

[20] Armido R Didonato and Alfred H Morris Jr. Algorithm 708: Significant digit computation of the incomplete beta function ratios. *ACM Transactions on Mathematical Software (TOMS)*, 18(3):360–373, 1992.

[21] John Duchi, Elad Hazan, and Yoram Singer. Adaptive subgradient methods for online learning and stochastic optimization. *The Journal of Machine Learning Research*, 12:2121–2159, 2011.

[22] Paul Ekman. *Telling Lies: Clues to Deceit in the Marketplace, Politics, and Marriage (Revised Edition)*. WW Norton & Company, 2009.

[23] Paul Ekman and Wallace V Friesen. *Manual for the facial action coding system*. Consulting Psychologists Press, 1978.

[24] Paul Ekman and Maureen O'Sullivan. Who can catch a liar? *American psychologist*, 46(9):913, 1991.

[25] Paul Ekman, Maureen O'Sullivan, and Mark G Frank. A few can catch a liar. *Psychological Science*, 10(3):263–266, 1999.

[26] Alhussein Fawzi, Omar Fawzi, and Pascal Frossard. Fundamental limits on adversarial robustness. ICML, 2015.

[27] Sally Floyd and Manfred Warmuth. Sample compression, learnability, and the Vapnik-Chervonenkis dimension. *Machine learning*, 21(3):269–304, 1995.

[28] Tommaso Fornaciari and Massimo Poesio. Automatic deception detection in Italian court cases. *Artificial intelligence and law*, 21(3):303–340, 2013.

[29] Vojtech Franc, Alexander Zien, and Bernhard Schölkopf. Support vector machines as probabilistic models. In *Proceedings of the 28th International Conference on Machine Learning (ICML-11)*, pages 665–672, 2011.

[30] M Frank, J Movellan, M Bartlett, and G Littleworth. RU-FACS-1 database. *Machine Perception Laboratory, UC San Diego*, 1, 2012.

[31] Stuart Geman, Elie Bienenstock, and René Doursat. Neural networks and the bias/variance dilemma. *Neural computation*, 4(1):1–58, 1992.

[32] Mathieu Germain, Karol Gregor, Iain Murray, and Hugo Larochelle. MADE: Masked Autoencoder for Distribution Estimation. *arXiv preprint arXiv:1502.03509*, 2015.

[33] Ian J Goodfellow, Jonathon Shlens, and Christian Szegedy. Explaining and harnessing adversarial examples. *arXiv preprint arXiv:1412.6572*, 2014.

[34] Ian J Goodfellow, David Warde-Farley, Mehdi Mirza, Aaron Courville, and Yoshua Bengio. Maxout networks. *arXiv preprint arXiv:1302.4389*, 2013.

[35] Yves Grandvalet, Johnny Mariéthoz, and Samy Bengio. Interpretation of SVMs with an application to unbalanced classification. In *Advances in Neural Information Processing Systems, NIPS 18*. Citeseer, 2005.

[36] Alex Graves. Practical variational inference for neural networks. In *Advances in Neural Information Processing Systems*, pages 2348–2356, 2011.

[37] Ralf Herbrich, Thore Graepel, and John Shawe-Taylor. Sparsity vs. Large Margins for Linear Classifiers. In *COLT*, pages 304–308, 2000.

[38] Geoffrey E Hinton, Nitish Srivastava, Alex Krizhevsky, Ilya Sutskever, and Ruslan R Salakhutdinov. Improving neural networks by preventing co-adaptation of feature detectors. *arXiv preprint arXiv:1207.0580*, 2012.

[39] Sepp Hochreiter, Yoshua Bengio, Paolo Frasconi, and Jürgen Schmidhuber. Gradient flow in recurrent nets: the difficulty of learning long-term dependencies, 2001.

[40] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural computation*, 9(8):1735–1780, 1997.

[41] Kurt Hornik. Approximation capabilities of multilayer feedforward networks. *Neural networks*, 4(2):251–257, 1991.

[42] David W Hosmer Jr and Stanley Lemeshow. *Applied logistic regression*. John Wiley & Sons, 2004.

[43] Sergey Ioffe and Christian Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. *arXiv preprint arXiv:1502.03167*, 2015.

[44] Uday Jain, Bozhao Tan, and Qi Li. Concealed knowledge identification using facial thermal imaging. In *Acoustics, Speech and Signal Processing (ICASSP), 2012 IEEE International Conference on*, pages 1677–1680. IEEE, 2012.

[45] Kam-Chuen Jim, C Lee Giles, and Bill G Horne. An analysis of noise in recurrent neural networks: convergence and generalization. *Neural Networks, IEEE Transactions on*, 7(6):1424–1438, 1996.

[46] Takeo Kanade, Jeffrey F Cohn, and Yingli Tian. Comprehensive database for facial expression analysis. In *Automatic Face and Gesture Recognition, 2000. Proceedings. Fourth IEEE International Conference on*, pages 46–53. IEEE, 2000.

[47] Andrej Karpathy and Li Fei-Fei. Deep visual-semantic alignments for generating image descriptions. *arXiv preprint arXiv:1412.2306*, 2014.

[48] Andrej Karpathy, George Toderici, Sachin Shetty, Tommy Leung, Rahul Sukthankar, and Li Fei-Fei. Large-scale video classification with convolutional neural networks. In *Computer Vision and Pattern Recognition (CVPR), 2014 IEEE Conference on*, pages 1725–1732. IEEE, 2014.

[49] Henry J Kelley. Gradient theory of optimal flight paths. *Ars Journal*, 30(10):947–954, 1960.

[50] Sander Koelstra, Maja Pantic, and Ioannis Patras. A dynamic texture-based approach to recognition of facial actions and their temporal models. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, 32(11):1940–1954, 2010.

[51] Andrei Nikolaevich Kolmogorov. On the representation of continuous functions of many variables by superposition of continuous functions of one variable and addition. *Amer. Math. Soc. Transl*, 28:55–59, 1963.

[52] Kishore Konda, Xavier Bouthillier, Roland Memisevic, and Pascal Vincent. Dropout as data augmentation. *arXiv preprint arXiv:1506.08700*, 2015.

[53] Alex Krizhevsky. Learning multiple layers of features from tiny images. *Master's thesis, University of Toronto*, 2009.

[54] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems*, pages 1097–1105, 2012.

[55] Ivan Laptev, Marcin Marszałek, Cordelia Schmid, and Benjamin Rozenfeld. Learning realistic human actions from movies. In *Computer Vision and Pattern Recognition, 2008. CVPR 2008. IEEE Conference on*, pages 1–8. IEEE, 2008.

[56] Quoc V Le. Building high-level features using large scale unsupervised learning. In *Acoustics, Speech and Signal Processing (ICASSP), 2013 IEEE International Conference on*, pages 8595–8598. IEEE, 2013.

[57] Quoc V Le, Will Y Zou, Serena Y Yeung, and Andrew Y Ng. Learning hierarchical invariant spatio-temporal features for action recognition with independent subspace analysis. In *Computer Vision and Pattern Recognition (CVPR), 2011 IEEE Conference on*, pages 3361–3368. IEEE, 2011.

[58] Yann LeCun. What is wrong with deep learning? Technical report, 2015.

[59] Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. Deep learning. *Nature*, 521(7553):436–444, 2015.

[60] Yann LeCun, Léon Bottou, Yoshua Bengio, and Patrick Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, 1998.

[61] Yann LeCun, Corinna Cortes, and Christopher JC Burges. The MNIST database of handwritten digits, 1998.

[62] Chen-Yu Lee, Saining Xie, Patrick Gallagher, Zhengyou Zhang, and Zhuowen Tu. Deeply-supervised nets. *arXiv preprint arXiv:1409.5185*, 2014.

[63] Gwen Littlewort, Jacob Whitehill, Tingfan Wu, Ian Fasel, Mark Frank, Javier Movellan, and Marian Bartlett. The computer expression recognition toolbox (CERT). In *Automatic Face & Gesture Recognition and Workshops (FG 2011), 2011 IEEE International Conference on*, pages 298–305. IEEE, 2011.

[64] Dong C Liu and Jorge Nocedal. On the limited memory BFGS method for large scale optimization. *Mathematical programming*, 45(1-3):503–528, 1989.

[65] Ulrike von Luxburg, Olivier Bousquet, and Bernhard Schölkopf. A compression approach to support vector model selection. *The Journal of Machine Learning Research*, 5:293–323, 2004.

[66] Dougal Maclaurin, David Duvenaud, and Ryan P Adams. Gradient-based Hyperparameter Optimization through Reversible Learning. *arXiv preprint arXiv:1502.03492*, 2015.

[67] Enes Makalic, Lloyd Allison, and David L Dowe. MML inference of single-layer neural networks. In *Proc. of the Third IASTED International Conference on Artificial Intelligence and Applications*, volume 1066, 2003.

[68] David Matsumoto, Hyi Sung Hwang, Lisa Skinner, and MG Frank. Evaluating truthfulness and detecting deception. *FBI Law Enforcement Bulletin*, 80:1–25, 2011.

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

[70] Marc Mehu, Marcello Mortillaro, Tanja Bänziger, and Klaus R Scherer. Reliable facial muscle activation enhances recognizability and credibility of emotional expression. *Emotion*, 12(4):701, 2012.

[71] Rada Mihalcea and Mihai Burzo. Towards multimodal deception detection–step 1: building a collection of deceptive videos. In *Proceedings of the 14th ACM international conference on Multimodal interaction*, pages 189–192. ACM, 2012.

[72] Rada Mihalcea, Verónica Pérez-Rosas, and Mihai Burzo. Automatic detection of deceit in verbal communication. In *Proceedings of the 15th ACM on International Conference on Multimodal Interaction*, pages 131–134. ACM, 2013.

[73] RGM Morris. DO Hebb: The Organization of Behavior, Wiley: New York; 1949. *Brain research bulletin*, 50(5):437, 1999.

[74] Vinod Nair and Geoffrey E Hinton. Rectified linear units improve restricted boltzmann machines. In *ICML 2010*, pages 807–814, 2010.

[75] Yurii Nesterov. A method of solving a convex programming problem with convergence rate O(1/k2). In *Soviet Mathematics Doklady*, volume 27, pages 372–376, 1983.

[76] Yuval Netzer, Tao Wang, Adam Coates, Alessandro Bissacco, Bo Wu, and Andrew Y Ng. Reading digits in natural images with unsupervised feature learning. In *NIPS workshop on deep learning and unsupervised feature learning*, volume 2011, page 5. Granada, Spain, 2011.

[77] Albert BJ Novikoff. On convergence proofs for perceptrons. Technical report, DTIC Document, 1963.

[78] Maja Pantic, Michel Valstar, Ron Rademaker, and Ludo Maat. Web-based database for facial expression analysis. In *Multimedia and Expo, 2005. ICME 2005. IEEE International Conference on*, pages 5–pp. IEEE, 2005.

[79] Verónica Pérez-Rosas, Rada Mihalcea, Alexis Narvaez, and Mihai Burzo. A Multimodal Dataset for Deception Detection. 2014.

[80] Tomas Pfister, Xiaobai Li, Guoying Zhao, and Matti Pietikainen. Recognising spontaneous facial micro-expressions. In *Computer Vision (ICCV), 2011 IEEE International Conference on*, pages 1449–1456. IEEE, 2011.

[81] Vu Pham, Théodore Bluche, Christopher Kermorvant, and Jérôme Louradour. Dropout improves recurrent neural networks for handwriting recognition. In *Frontiers in Handwriting Recognition (ICFHR), 2014 14th International Conference on*, pages 285–290. IEEE, 2014.

[82] John Platt et al. Probabilistic outputs for support vector machines and comparisons to regularized likelihood methods. *Advances in large margin classifiers*, 10(3):61–74, 1999.

[83] Ben Poole, Jascha Sohl-Dickstein, and Surya Ganguli. Analyzing noise in autoencoders and deep networks. *arXiv preprint arXiv:1406.1831*, 2014.

[84] Stephen Porter, Leanne ten Brinke, and Brendan Wallace. Secrets and lies: Involuntary leakage in deceptive facial expressions as a function of emotional intensity. *Journal of Nonverbal Behavior*, 36(1):23–37, 2012.

[85] Salah Rifai, Pascal Vincent, Xavier Muller, Xavier Glorot, and Yoshua Bengio. Contractive auto-encoders: Explicit invariance during feature extraction. In *ICML 2011*, pages 833–840, 2011.

[86] Daniel Rueckert, Luke I Sonoda, Carmel Hayes, Derek LG Hill, Martin O Leach, and David J Hawkes. Nonrigid registration using free-form deformations: application to breast MR images. *Medical Imaging, IEEE Transactions on*, 18(8):712–721, 1999.

[87] Andrew M Saxe, James L McClelland, and Surya Ganguli. Exact solutions to the nonlinear dynamics of learning in deep linear neural networks. *arXiv preprint arXiv:1312.6120*, 2013.

[88] Bernhard Schölkopf, Alexander Smola, and Klaus-Robert Müller. Kernel principal component analysis. pages 583–588, 1997.

[89] Patrice Y Simard, Yann A LeCun, John S Denker, and Bernard Victorri. Transformation invariance in pattern recognition–tangent distance and tangent propagation. In *Neural networks: tricks of the trade, pages=239–274*. Springer, 1998.

[90] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556*, 2014.

[91] Thomas E Slowe and Venu Govindaraju. Automatic deceit indication through reliable facial expressions. In *Automatic Identification Advanced Technologies, 2007 IEEE Workshop on*, pages 87–92. IEEE, 2007.

[92] Jasper Snoek, Hugo Larochelle, and Ryan P Adams. Practical Bayesian optimization of machine learning algorithms. In *Advances in Neural Information Processing Systems*, pages 2951–2959, 2012.

[93] Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. Dropout: A simple way to prevent neural networks from overfitting. *The Journal of Machine Learning Research*, 15(1):1929–1958, 2014.

[94] Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andrew Rabinovich. Going deeper with convolutions. *arXiv preprint arXiv:1409.4842*, 2014.

[95] Christian Szegedy, Wojciech Zaremba, Ilya Sutskever, Joan Bruna, Dumitru Erhan, Ian Goodfellow, and Rob Fergus. Intriguing properties of neural networks. *arXiv preprint arXiv:1312.6199*, 2013.

[96] Yichuan Tang and Chris Eliasmith. Deep networks for robust visual recognition. In *Proceedings of the 27th International Conference on Machine Learning (ICML-10)*, pages 1055–1062, 2010.

[97] Graham W Taylor, Rob Fergus, Yann LeCun, and Christoph Bregler. Convolutional learning of spatio-temporal features. In *Computer Vision–ECCV 2010*, pages 140–153. Springer, 2010.

[98] Robert Tibshirani. Regression shrinkage and selection via the LASSO. *Journal of the Royal Statistical Society. Series B (Methodological)*, pages 267–288, 1996.

[99] Andrey Nikolaevich Tikhonov and Vasiliy Yakovlevich Arsenin. *Solutions of ill-posed problems*. Vh Winston, 1977.

[100] Helge Tverberg. A generalization of Radon's theorem. *J. London Math. Soc*, 41(1):123–128, 1966.

[101] Vladimir Vapnik. *The nature of statistical learning theory*. Springer Science & Business Media, 2013.

[102] Vladimir Vapnik and Olivier Chapelle. Bounds on error expectation for support vector machines. *Neural computation*, 12(9):2013–2036, 2000.

[103] Vladimir N Vapnik and A Ya Chervonenkis. On the uniform convergence of relative frequencies of events to their probabilities. *Theory of Probability & Its Applications*, 16(2):264–280, 1971.

[104] Paul Viola and Michael Jones. Rapid object detection using a boosted cascade of simple features. In *Computer Vision and Pattern Recognition, 2001. CVPR 2001. Proceedings of the 2001 IEEE Computer Society Conference on*, volume 1, pages I–511. IEEE, 2001.

[105] John Von Neumann. The general and logical theory of automata. *Cerebral mechanisms in behavior*, pages 1–41, 1951.

[106] John Von Neumann. Probabilistic logics and the synthesis of reliable organisms from unreliable components. *Automata studies*, 34:43–98, 1956.

[107] Aldert Vrij. *Detecting lies and deceit: The psychology of lying and implications for professional practice*. Wiley, 2000.

[108] Aldert Vrij. *Detecting lies and deceit: Pitfalls and opportunities*. John Wiley & Sons, 2008.

[109] Sida Wang and Christopher Manning. Fast dropout training. In *Proceedings of the 30th International Conference on Machine Learning (ICML-13)*, pages 118–126, 2013.

[110] Lara Warmelink, Aldert Vrij, Samantha Mann, Sharon Leal, Dave Forrester, and Ronald P Fisher. Thermal imaging as a lie detection tool at airports. *Law and human behavior*, 35(1):40, 2011.

[111] Gemma Warren, Elizabeth Schertler, and Peter Bull. Detecting deception from emotional and unemotional cues. *Journal of Nonverbal Behavior*, 33(1):59–69, 2009.

[112] Paul Werbos. Beyond regression: New tools for prediction and analysis in the behavioral sciences. *PhD thesis, Harvard University*, 1974.

[113] Paul J Werbos. Backpropagation through time: what it does and how to do it. *Proceedings of the IEEE*, 78(10):1550–1560, 1990.

[114] Bernard Widrow. Generalization and information storage in network of ADALINE "neurons". *Self-organizing systems-1962*, pages 435–462, 1962.

[115] Bernard Widrow, Marcian E Hoff, et al. Adaptive switching circuits. 1960.

[116] Bernard Widrow, Michael Lehr, et al. 30 years of adaptive neural networks: perceptron, madaline, and backpropagation. *Proceedings of the IEEE*, 78(9):1415–1442, 1990.

[117] Neha Yadav, Anupam Yadav, and Manoj Kumar. *An Introduction to Neural Network Methods for Differential Equations*. Springer, 2015.

[118] Wen-Jing Yan, Su-Jing Wang, Yong-Jin Liu, Qi Wu, and Xiaolan Fu. For micro-expression recognition: Database and suggestions. *Neurocomputing*, 136:82–87, 2014.

[119] Wen-Jing Yan, Qi Wu, Yong-Jin Liu, Su-Jing Wang, and Xiaolan Fu. Casme database: a dataset of spontaneous micro-expressions collected from neutralized faces. In *Automatic Face and Gesture Recognition (FG), 2013 10th IEEE International Conference and Workshops on*, pages 1–7. IEEE, 2013.

[120] Wojciech Zaremba, Ilya Sutskever, and Oriol Vinyals. Recurrent neural network regularization. *arXiv preprint arXiv:1409.2329*, 2014.

[121] Matthew D Zeiler and Rob Fergus. Stochastic pooling for regularization of deep convolutional neural networks. *arXiv preprint arXiv:1301.3557*, 2013.

[122] Lei Zhang, Yan Tong, and Qiang Ji. Active image labeling and its application to facial action labeling. In *Computer Vision–ECCV 2008*, pages 706–719. Springer, 2008.

[123] Zhi Zhang, Vartika Singh, Thomas E Slowe, Sergey Tulyakov, and Venugopal Govindaraju. Real-time automatic deceit detection from involuntary facial expressions. In *Computer Vision and Pattern Recognition, 2007. CVPR'07. IEEE Conference on*, pages 1–6. IEEE, 2007.

[124] Guoying Zhao and Matti Pietikainen. Dynamic texture recognition using local binary patterns with an application to facial expressions. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, 29(6):915–928, 2007.

[125] Miron Zuckerman, Bella M DePaulo, and Robert Rosenthal. Verbal and nonverbal communication of deception. *Advances in experimental social psychology*, 14:1–59, 1981.

Author/s:
Demyanov, Sergey

Title:
Regularization methods for neural networks and related models

Date:
2015

Persistent Link:
http://hdl.handle.net/11343/57198