Learning in Compressed Space

Alexander Fabisch[a], Yohannes Kassahun[a], Hendrik Wöhrle[b], Frank Kirchner[a,b]

[a] *University of Bremen, Fachbereich 3 - Mathematik und Informatik, Postfach 330 440, 28334 Bremen, Germany*
[b] *Robotics Innovation Center, German Research Center for Artificial Intelligence (DFKI), Robert-Hooke-Str. 5, 28359 Bremen, Germany*

Abstract

We examine two methods which are used to deal with complex machine learning problems: compressed sensing and model compression. We discuss both methods in the context of feed-forward artificial neural networks and develop the backpropagation method in compressed parameter space. We further show that compressing the weights of a layer of a multilayer perceptron is equivalent to compressing the input of the layer. Based on this theoretical framework, we will use orthogonal functions and especially random projections for compression and perform experiments in supervised and reinforcement learning to demonstrate that the presented methods reduce training time significantly.

Keywords: model compression, compressed sensing, artificial neural networks, supervised learning, reinforcement learning

1. Introduction

Artificial intelligence is facing real world problems and thus machine learning problems become more and more complex. Bengio and Lecun [5] state that "a long-term goal of machine learning research is to produce methods that will enable artificially intelligent agents capable of learning complex

Email addresses: `afabisch@informatik.uni-bremen.de` (Alexander Fabisch), `kassahun@informatik.uni-bremen.de` (Yohannes Kassahun), `Hendrik.Woehrle@dfki.de` (Hendrik Wöhrle), `Frank.Kirchner@dfki.de` (Frank Kirchner)

behaviors with minimal human intervention and prior knowledge". As a result, even more effort is shifted from human to machine.

Complex machine learning problems comprise learning complex behaviors like visual and auditory perception and natural language processing as mentioned by Bengio and Lecun [5] as well as dealing with highly noisy data such as electroencephalography (EEG) signals and learning behavior and perception for complex robots with many actuators and sensors. Examples for complex robots that have many actuators and sensors are ASGUARD [19], SCORPION [47] and SpaceClimber [3, 42]. A common characteristic of complex problems is an associated large amount of data, which includes a large input space dimension and/or a large training set. For example, vision problems usually have hundreds to millions of input components as well as thousands of training examples to cover all possible distortions and a typical brain-computer interface (BCI) [51] can sample 128 channels with 5 kHz.

Complex problems usually result in long training times. We will now consider ways that reduce the training time. There are numerous ways of dealing with large input spaces that reduce the training time. Examples include methods of feature selection [23], dimensionality reduction or feature extraction. These methods require in most cases at least some domain specific knowledge and a manually designed preprocessing flow. But developments that reduce the need for human expertise and intervention such as convolutional neural networks (CNNs) [35] or deep belief neural networks (DBNs) [4] exist as well. The first layers of these kinds of neural networks can be seen as trainable feature extractors, which can be used to cope with large input spaces. This way of dealing with machine learning problems is better because information cannot be lost unintentionally when (almost) unprocessed data is used as input for the learning algorithm.

Another way of avoiding unintentional loss of information is using conventional machine learning algorithms without built-in preprocessing on raw data. But this is computationally more challenging and requires lots of optimizations, in particular during the training. In this article we want to present novel approaches that can deal with complex machine learning problems. In particular, we want to show how to reduce the time needed to train feed-forward neural networks.

We consider only multilayer perceptrons (MLPs) here to demonstrate two ways of simplifying machine learning optimization problems: model compression and data compression. Advantages of MLPs are that they are universal function approximators [26] and they are simple in concept, well-known and widely used. Since MLP layers are contained in CNNs and the backpropagation method is also used in DBNs, we think that this restriction is justifiable. In addition, the presented concepts can be extended to many other learning algorithms. One of these learning algorithms is the support vector machine (SVM) [10, 49]. We can combine SVMs and data compression easily and use SVMs to compare them to MLPs in this article.

The source code for most of the experiments we present here is available online at `https://github.com/AlexanderFabisch/OpenANN`.

2. Related Work

We will first give a short overview of the foundations of our work, which include two research branches. In addition, we will discuss related ideas.

We call the first research branch “model compression”. Schmidhuber [45, 46] developed a universal *network encoding language* (NEL) to represent neural networks in a compressed form for supervised learning. This was motivated by the search for the best generalizing network, because the simplest hypothesis that fits the training instances should give the best generalization for the latent function. Since the NEL is not continuous it was not possible to use efficient optimization algorithms for this compressed representation. This was the reason for Koutník et al. [31, 32] to develop a continuous representation, where the weights of a recurrent neural network with fixed topology are represented by coefficients of an inverse discrete cosine transform. The focus of their work was reinforcement learning [48]. Therefore, they presented experiments with the evolutionary optimization algorithm CoSyNE [22]. The main goal in this research is the increase of sample efficiency, that is the reduction of episodes needed to learn a good or successful policy. We will present a very similar approach in this article. However, there will be some differences:

- We will focus on feed-forward neural networks, and hence extend the standard backpropagation procedure for learning in compressed space.
- Not only a combination of orthogonal cosine functions, but any kind of orthogonal functions and even randomly generated values can be used to generate the weights. Hence, we will provide a more general framework for model compression.

- Koutník et al. [31, 32] compress all weights of a neural network with the same coefficients, although they already mention that it would be better to have a distinct set of coefficients for each neuron. We will use a distinct set of coefficients for each neuron to generate all incoming weights.

The second research branch is the so called "compressed sensing" [13, 18]. In compressed sensing, sparse or compressible data is examined. It is possible to compress compressible data through *random projections* and reconstruct it with high probability by solving an optimization problem. In combination with machine learning it can be used as a preprocessing method that requires almost no prior knowledge. In particular, the method does not need to compute any features from the training set. Compressed sensing has for example been combined with support vector machines by Calderbank et al. [12] and least squares regression by Maillard and Munos [38]. In these cases the reconstruction is not necessary. It is instead assumed that it is possible to distinguish the instances in compressed space because we could reconstruct the original data with high probability. This is true because random projections approximately preserve distances between instances [6]. Random projections are a very powerful technique to overcome the curse of dimensionality when approximate solutions are sufficient. This has for example been proven for similarity search as well (see locality-sensitive hashing [21, 27]).

We have already shown in a previous paper [30] that compressed sensing and compressing the weights of a neural network that has no hidden layer (single layer perceptron, SLP) are mathematically equivalent. As a contribution to the state of the art, we will show that compressing the weight matrix of any fully connected layer is equivalent to compressing the input to that layer.

Our goal is to reduce training time by reducing the number of parameters that we have to optimize for a neural network. There are other approaches in machine learning that have the same goals and could be combined with the methods we present here. Two ways to do this with neural networks are weight sharing and sparse connections, which are actually forms of model compression. Prominent examples that combine both methods are CNNs [35]. These have been particularly successful in recognizing objects in images such as traffic signs [16]. All kinds of CNNs consist of at least one convolutional layer. A very simple example is shown in Figure 1. In this convolutional layer, we have a two-dimensional input with 3×3 entries and

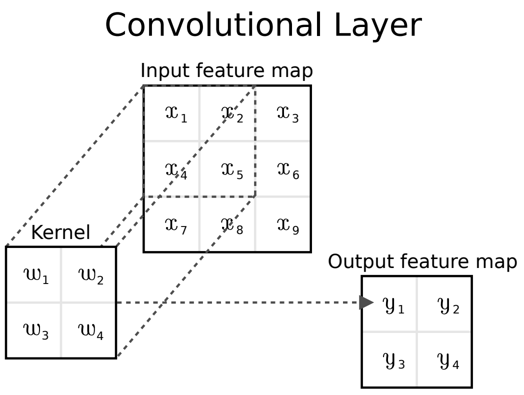

Figure 1: Convolutional layer of a convolutional neural network. The input feature map is convolved with a parametrizable kernel to create the output. For example, the first entry of the output feature map is $y_1 = w_1x_1 + w_2x_2 + w_3x_4 + w_4x_5$.

a two-dimensional output with 2×2 entries. These are called feature maps. The input is convolved with a 2×2 kernel to obtain the output. We can construct a fully connected layer, that computes the same output through

$$\begin{pmatrix} y_1 \\ y_2 \\ y_3 \\ y_4 \end{pmatrix} = \boldsymbol{W} \cdot \begin{pmatrix} x_1 \\ \vdots \\ x_9 \end{pmatrix}, \tag{1}$$

where $\boldsymbol{W}$ is the weight matrix

$$\boldsymbol{W} = \begin{pmatrix} w_1 & w_2 & 0 & w_3 & w_4 & 0 & 0 & 0 & 0 \\ 0 & w_1 & w_2 & 0 & w_3 & w_4 & 0 & 0 & 0 \\ 0 & 0 & 0 & w_1 & w_2 & 0 & w_3 & w_4 & 0 \\ 0 & 0 & 0 & 0 & w_1 & w_2 & 0 & w_3 & w_4 \end{pmatrix}. \tag{2}$$

It is obvious that $\boldsymbol{W}$ is sparse and the weights $w_1, \ldots, w_4$ occur in more than one row in this matrix, that is, they are shared among neurons. As a result, we can say that the weight matrix $\boldsymbol{W}$ is generated by a transformation of the convolution kernel. We can make a similar statement about any kind of convolutional layer: the weight matrix of a convolutional layer is generated from less parameters by a transformation. Why this is a form of model compression will become clear at the end of Section 4.1.

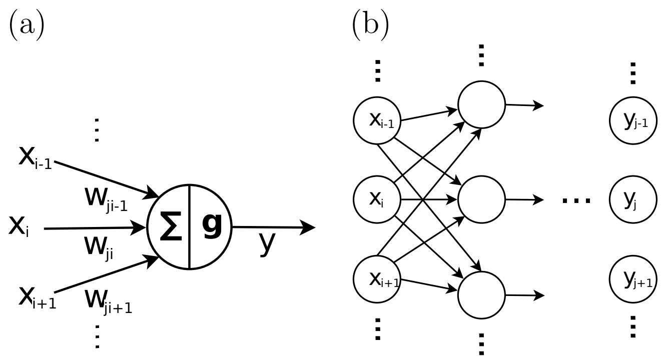

Figure 2: (a) Artificial neuron. (b) Multilayer perceptron.

3. Basics

Before we start with the discussion of model compression, we want to summarize some basic knowledge about neural networks, introduce notation we will use throughout this article and define what is meant by reinforcement learning for our purpose.

3.1. Artificial Neural Networks

In supervised or reinforcement learning we want to build a model of an unknown function $\boldsymbol{y} = f(\boldsymbol{x})$ with $\boldsymbol{x} \in \mathbb{R}^D, \boldsymbol{y} \in \mathbb{R}^F$. When we talk about a *model* in this article, we mean a feed-forward artificial neural network, which is a parametrizable function $\boldsymbol{y} = \hat{f}(\boldsymbol{x}, \boldsymbol{w})$, where $\boldsymbol{w} \in \mathbb{R}^K$ is a *weight vector*.

The basic module of an artificial neural network (ANN) is the artificial neuron (see Figure 2 (a)). It computes a weighted sum of its inputs x_i, which is called *activation*, and applies an activation function g to generate the output y. These neurons can be connected and arranged in layers to form a multilayer perceptron, where consecutive layers are fully connected (see Figure 2 (b)). The output of the model $\boldsymbol{y}$ will be calculated through forward propagation, that is the neurons will sequentially calculate their outputs and the outputs of the last layer will generate the model's output.

In order to fit the function f, we usually modify the model parameters $\boldsymbol{w}$ directly, such that they minimize the error on the training set in supervised learning or maximize the return in reinforcement learning. That is, in supervised learning we use an optimization algorithm to minimize an error function $E(\boldsymbol{w})$. Fast optimization algorithms (see for example Table 1) usually require the gradient of the error function $\nabla E(\boldsymbol{w}) = \sum_{n=1}^{N} \nabla E_n(\boldsymbol{w})$, where E_n is the error of training instance n.

In order to calculate the gradient, we use the backpropagation procedure [43]. This requires the calculation of a δ_j for each neuron j. This quantity can be interpreted as the contribution of neuron j to the error. For each neuron j in the output layer this is calculated as

$$\delta_j = \frac{\partial E_n}{\partial a_j} = g'(a_j)\frac{\partial E_n}{\partial y_j}, \tag{3}$$

where g is the activation function of neuron j, a_j is its activation and y_j its output. Then we can *backpropagate* these δ's in order to calculate the δ's of lower layers through

$$\delta_j = \frac{\partial E_n}{\partial a_j} = g'(a_j)\sum_k w_{kj}\delta_k, \tag{4}$$

where neuron k is in the layer above j. Finally, we can compute the derivative of the error with respect to each weight

$$\frac{\partial E_n}{\partial w_{ji}} = \delta_j z_i, \tag{5}$$

where z_i is the output of neuron i.

3.2. Design of Artificial Neural Networks

Conventional Artificial Neural Networks like MLPs usually have a disadvantage: it is hard to apply them to problems because their architecture has to be determined in advance. Here, the “architecture” subsumes the number of hidden layers, the number of neurons, the type of activation functions, the error function, etc. In our experiments, we usually followed the advices of Bishop [7], Sarle [44] and LeCun et al. [36] and extracted the following rules:

- Keep the architecture as simple as possible. We start from a single layer perceptron (SLP) and will only add layers and/or neurons if that improves the performance the neural network in terms of predictions and/or time.

- In all hidden layers we use the activation function *tanh*.

- For two-class classification problems, we use *tanh* in the output layer. For regression, we use the identity $g(x) = x$ in the output layer. In both cases we use the error function sum of squared errors (SSE). For more than two classes, we use softmax activation functions and cross entropy error function. Some reinforcement learning problems require more tailored solutions, for example because the output has to be within a given interval.

In this article, we use the following notation to describe the architecture of an MLP:

$$J^{(0)}\text{-}J^{(1)}\text{-}\ldots\text{-}J^{(l)}\text{-}\ldots\text{-}J^{(P)},$$

where $J^{(l)}$ is the number of neurons in layer l and $P+1$ is the total number of layers, including input layer, hidden layers and output layer. For example, 2-20-20-1 means that there are 2 inputs, 20 hidden nodes in the first hidden layer, 20 hidden nodes in the second hidden layer and 1 output node and there is usually an additional bias in every layer.

3.3. Reinforcement Learning

When we talk about reinforcement learning in this article, we will focus on direct policy search. Usually, reinforcement learning requires the approximation of value functions that indicate how good it is for an agent to be in a given state and then we can infer policies from these value functions [48]. In direct policy search, the policies are learned directly. In this article, we use an optimization algorithm that directly adjusts the parameters of the policy to maximize the return of an episode and the policy is represented by a neural network.

4. Learning in Compressed Space

In this section we first describe a novel generalized framework for model compression of artificial neural networks and present its advantages. In particular, we derive the extension of the backpropagation method for neural networks. Afterwards we give an introduction to data compression in the context of compressed sensing. Finally, we prove and discuss the equivalence of model compression and data compression.

4.1. Model Compression

We can compress the model $\hat{f}(\boldsymbol{x}, \boldsymbol{w})$ by representing $\boldsymbol{w} \in \mathbb{R}^K$ with less than K parameters.

Definition 1. Let $w_{ji},\ i \in \{1, \ldots, I_j\}$ be the weights of a neuron j as depicted in Figure 2 (a). The weight w_{ji} is generated from compressed parameter space by

$$w_{ji} = \sum_{m=1}^{M_j} \alpha_{jm} \phi_{mi}, \tag{6}$$

where M_j is the number of parameters $\alpha_{j1}, \ldots, \alpha_{jM_j}$ to be optimized and ϕ_{mi} can be

- sampled from pairwise orthogonal functions, that is $\phi_{mi} = \phi_m(t_i)$ where for all $k, l \in \{1, \ldots, M_j\}$

  $$\int_0^1 \phi_k(t)\phi_l(t)\ dt = 0, \tag{7}$$

  and t_i is defined as

  $$t_i = \frac{i-1}{I_j - 1}, \tag{8}$$

  for example, the set $\{\phi_0(t_i) = \cos(0\pi t_i), \phi_1(t_i) = \cos(1\pi t_i), \phi_2(t_i) = \cos(2\pi t_i), \ldots\}$ contains orthogonal cosine functions,

- or sampled from a random distribution that we discuss in Section 4.3.

In order to compress the model, $\sum_j M_j = L < K$ must be satisfied. That is, the weights of at least one neuron have to be represented by less parameters than the number of weights of the neuron.

The quantity $\boldsymbol{\alpha}$ is a vector that contains all parameters that are required to generate the weights of an ANN. For supervised learning, we can extend the standard backpropagation procedure to calculate the gradient of $E(\boldsymbol{\alpha})$.

Theorem 1. *Given the partial derivative of the error E_n with respect to the weight w_{ji}, the partial derivative of the error E_n with respect to the parameter α_{jm} can be calculated by*

$$\frac{\partial E_n}{\partial \alpha_{jm}} = \sum_i \frac{\partial E_n}{\partial w_{ji}} \phi_{mi}. \tag{9}$$

Proof. We apply the chain rule for partial derivatives.

$$\frac{\partial E_n}{\partial \alpha_{jm}} = \sum_i \frac{\partial E_n}{\partial w_{ji}} \frac{\partial w_{ji}}{\partial \alpha_{jm}} \tag{10}$$

Using Equation (6), we can write

$$\begin{aligned} \frac{\partial E_n}{\partial \alpha_{jm}} &= \sum_i \frac{\partial E_n}{\partial w_{ji}} \frac{\partial}{\partial \alpha_{jm}} \left(\sum_{m'=1}^{M_j} \alpha_{jm'} \phi_{m'i} \right) && (11) \\ &= \sum_i \frac{\partial E_n}{\partial w_{ji}} \phi_{mi}. \quad \square && (12) \end{aligned}$$

Since we focus on MLPs, the calculation of the activation for each layer l is usually done by a matrix-vector multiplication

$$\boldsymbol{a}^{(l)} = \boldsymbol{W}^{(l)} \boldsymbol{x}^{(l)}, \tag{13}$$

where $\boldsymbol{W}^{(l)} \in \mathbb{R}^{J^{(l)} \times I^{(l)}}$ is the weight matrix of layer l with its components being $(\boldsymbol{W}^{(l)})_{ji} = w_{ji}^{(l)}$, $\boldsymbol{x}^{(l)} \in \mathbb{R}^{I^{(l)}}$ is the input of layer l and $\boldsymbol{a}^{(l)} \in \mathbb{R}^{J^{(l)}}$ contains the activations of each neuron $j \in \{1, \ldots, J^{(l)}\}$. If we use a bias in this layer, $\boldsymbol{x}^{(l)}$ will be the output of the previous layer with an additional component that is always 1. For MLPs we can simplify the model compression.

Theorem 2. *If we use the same number of parameters $M^{(l)}$ for all neurons in layer l, the weight matrix of layer l is generated by*

$$\boldsymbol{W}^{(l)} = \boldsymbol{\alpha}^{(l)} \boldsymbol{\Phi}^{(l)}, \tag{14}$$

where $\boldsymbol{\alpha}^{(l)} \in \mathbb{R}^{J^{(l)} \times M^{(l)}}$ is a parameter matrix with the components $(\boldsymbol{\alpha}^{(l)})_{jm} = \alpha_{jm}^{(l)}$ and $\boldsymbol{\Phi}^{(l)} \in \mathbb{R}^{M^{(l)} \times I^{(l)}}$ is a constant matrix whose components are generated by $(\boldsymbol{\Phi}^{(l)})_{mi} = \phi_{mi}$.

Proof. When we look at the components of the matrices in Equation (14)

$$\begin{aligned} & \begin{pmatrix} \alpha_{11}^{(l)} & \cdots & \alpha_{1M^{(l)}}^{(l)} \\ \vdots & \ddots & \vdots \\ \alpha_{J^{(l)}1}^{(l)} & \cdots & \alpha_{J^{(l)}M^{(l)}}^{(l)} \end{pmatrix} \cdot \begin{pmatrix} \phi_{11} & \cdots & \phi_{1I^{(l)}} \\ \vdots & \ddots & \vdots \\ \phi_{M^{(l)}1} & \cdots & \phi_{M^{(l)}I^{(l)}} \end{pmatrix} && (15) \\ &= \begin{pmatrix} w_{11}^{(l)} & \cdots & w_{1I^{(l)}}^{(l)} \\ \vdots & \ddots & \vdots \\ w_{J^{(l)}1}^{(l)} & \cdots & w_{J^{(l)}I^{(l)}}^{(l)} \end{pmatrix}, && (16) \end{aligned}$$

we can see that the weight $w_{ji}^{(l)}$ is generated by Equation (6), since

$$\begin{pmatrix} \alpha_{j1}^{(l)} & \dots & \alpha_{jM^{(l)}}^{(l)} \end{pmatrix} \cdot \begin{pmatrix} \phi_{1i} \\ \vdots \\ \phi_{M^{(l)}i} \end{pmatrix} = \sum_{m=1}^{M^{(l)}} \alpha_{jm}^{(l)} \phi_{mi}. \quad \square \tag{17}$$

Theorem 3. *If we use backpropagation, we can construct a matrix* $\boldsymbol{V}^{(l)} \in \mathbb{R}^{J^{(l)} \times I^{(l)}}$*, where* $(\boldsymbol{V}^{(l)})_{ji} = \frac{\partial E_n}{\partial (\boldsymbol{W}^{(l)})_{ji}}$ *and use this matrix to form another matrix* $\boldsymbol{\Omega}^{(l)} \in \mathbb{R}^{J^{(l)} \times M^{(l)}}$*, where* $(\boldsymbol{\Omega}^{(l)})_{jm} = \frac{\partial E_n}{\partial (\boldsymbol{\alpha}^{(l)})_{jm}}$ *with*

$$\boldsymbol{\Omega}^{(l)} = \boldsymbol{V}^{(l)} (\boldsymbol{\Phi}^{(l)})^T. \tag{18}$$

The proof for this theorem is straight forward and only requires to write Equation (9) in matrix form. Thus, we will omit it here.

This method of model compression requires a transformation of the parameter matrix $\boldsymbol{\alpha}^{(l)}$ to generate the weight matrix $\boldsymbol{W}^{(l)}$. This is in principle similar to what is done in convolutional layers of CNNs implicitely. In convolutional layers, we could generate $\boldsymbol{W}^{(l)}$ from the convolution kernel by a transformation.

In Section 3.2 we describe how we abbreviate the description of an architecture of an ANN. Similarly, we describe the compression:

$$M^{(1)}\text{-}\dots\text{-}M^{(l)}\text{-}\dots\text{-}M^{(P)},$$

where $M^{(l)}$ is the number of parameters that is used to generate the weights of each neuron in layer l. For example, 3-6-3 means that we compress the weights of the first hidden layer with 3 parameters per neuron, in the second hidden layer with 6 parameters per neuron and in the output layer with 3 parameters per neuron.

4.2. Advantages of Model Compression

There are at least three ways to reduce the training time of neural networks:

Option 1: We can reduce the time that is needed for forward propagation and backpropagation.

Option 2: We can reduce the time that the optimization algorithm needs for each iteration. Here we have to exclude the time of forward propagation and backpropagation.

Algorithm	Time complexity
Newton Algorithm	$O(L^3)$
Gauß-Newton	$O(L^3)$
Levenberg-Marquardt	$O(L^3)$
BFGS	$O(L^2)$
Conjugate Gradient	$O(L)$

Table 1: Time complexity of each iteration for several gradient based optimization algorithms according to LeCun et al. [36]. The number of parameters that will be optimized is denoted by L.

Option 3: We can reduce the number of iterations an optimization algorithm needs to converge.

Reducing the number of model parameters has different effects in supervised learning and reinforcement learning with direct policy search.

When we look at the extended backpropagation that is required when we use this model compression method for supervised learning, we can see that it only adds more computational cost. In addition, we have to generate the weights anew, when we adjust the parameters. Thus, we clearly do not use option 1. We even cannot assure that the number of iterations is reduced (option 3), because an indirect weight optimization can even make the optimization harder. Nevertheless, the overall learning process will speed up, when we use complex optimization algorithms that are in $O(L^2)$ or even $O(L^3)$, where L is the number of parameters that have to be optimized, and we can reduce the number of parameters in comparison to the number of weights sufficiently. Then each iteration of the optimization algorithm requires only a fraction of time. Interestingly, some of the best optimization algorithms for batch learning have high time complexity in each optimization step (see Table 1).

We think that the computational cost is not important for reinforcement learning. This is because the goal of reinforcement learning is to deal with complex simulated environments that are computationally expensive or with the real world. In the real world we want to minimize the possibility to wear or break real systems. Thus, we have to minimize the number of function evaluations an optimization algorithm needs. Each function evaluation in this setup is called *episode*. The derivative-free optimization algorithm we use here for direct policy search is CMA-ES [24]. To our knowledge there is no convergence proof for CMA-ES yet, but our empirical results show that the convergence speed depends on the search space dimension L. This is the reason why the reduction of parameters will speed up the learning process.

4.3. Data Compression

Compressed sensing can be used for data compression. Baraniuk et al. [2] summarize

> "In *Compressed Sensing* (CS) [..], a random projection of a high-dimensional but sparse or compressible signal vector onto a lower-dimensional space has been shown, with high probability, to contain enough information to enable signal reconstruction with small or zero error."

In specific terms that means one of the key equations in CS [13, 18] is

$$\boldsymbol{y} = \boldsymbol{\Phi x}, \tag{19}$$

where $\boldsymbol{x} \in \mathbb{R}^N$ is sparse or compressible, $\boldsymbol{y} \in \mathbb{R}^M$ and $M < N$. $\boldsymbol{\Phi} \in \mathbb{R}^{M\times N}$ is generated randomly and has to satisfy the *restricted isometry property* (RIP) [14]. In this setting, we could approximately reconstruct $\boldsymbol{x}$ from $\boldsymbol{y}$ with high probability. However, this is not required when we use CS as a preprocessing method for machine learning.

According to Candès and Romberg [13], a signal $\boldsymbol{x}$ is said to be compressible if the reordered $\boldsymbol{\Psi}$-coefficients $\boldsymbol{\nu}$, $\boldsymbol{x} = \boldsymbol{\Psi\nu}$, in decreasing order of magnitude decay like a power law. That is, in the sequence

$$|\boldsymbol{\nu}|_{(1)} \geq |\boldsymbol{\nu}|_{(2)} \geq \ldots \geq |\boldsymbol{\nu}|_{(N)}, \tag{20}$$

the nth entry obeys

$$|\boldsymbol{\nu}|_{(n)} \leq c \cdot n^{-s} \tag{21}$$

for some constants c and $s \geq 1$. That means it will be sufficient if the signal is approximately sparse in *some* orthogonal basis $\boldsymbol{\Psi}$.

Random distributions we use to generate the matrix $\boldsymbol{\Phi}$ have to satisfy the RIP. Candès and Tao [14] originally define the restricted isometry property of a matrix $\boldsymbol{\Phi}$ that has *restricted isometry constants* δ_M. These constants limit the deviation of the norm, when we project any vector $\boldsymbol{x}$ on any M-dimensional subspace by multiplying it with a submatrix $\boldsymbol{\Phi}_M$ of $\boldsymbol{\Phi}$, that is

$$(1-\delta_M)||\boldsymbol{x}||^2 \leq ||\boldsymbol{\Phi}_M\boldsymbol{x}||^2 \leq (1+\delta_M)||\boldsymbol{x}||^2. \tag{22}$$

This definition is then used to prove that sparse data can be reconstructed after compression. In addition, they show that there exist matrices with good restricted isometry constants.

Baraniuk et al. [2] show for example that

$$\boldsymbol{\Phi}_{mi} \sim \mathcal{N}(0, \frac{1}{M_j}), \tag{23}$$

$$\boldsymbol{\Phi}_{mi} = \begin{cases} \frac{+1}{\sqrt{M_j}} & \text{with probability } \frac{1}{2} \\ \frac{-1}{\sqrt{M_j}} & \text{with probability } \frac{1}{2} \end{cases}, \tag{24}$$

$$\text{and } \boldsymbol{\Phi}_{mi} = \begin{cases} \frac{+1}{\sqrt{M_j}} & \text{with probability } \frac{1}{6} \\ 0 & \text{with probability } \frac{2}{3} \\ \frac{-1}{\sqrt{M_j}} & \text{with probability } \frac{1}{6} \end{cases} \tag{25}$$

satisfy the RIP.

Another property of these random projections is that they approximately preserve distances between instances because they satisfy the Johnson-Lindenstrauß lemma [28] according to Baraniuk et al. [2]. In other words, random projections preserve similarities of data vectors. This property is especially useful for machine learning algorithms, because their ability to generalize is based upon the fact that similar inputs have similar outputs. We can regard compressed sensing as a preprocessing method that requires almost no domain specific knowledge. We only have to know that the data is compressible.

4.4. Equivalence of Model Compression and Data Compression

Theorem 4. *Compressing the weight matrix $\boldsymbol{W}^{(l)}$ of layer l is equivalent to compressing the input $\boldsymbol{x}^{(l)}$ of this layer.*

Proof. Let us assume that we generate $\boldsymbol{W}^{(l)}$ according to Equation (14). Then we can write the calculation of the activations in layer l as

$$\begin{aligned} \boldsymbol{W}^{(l)}\boldsymbol{x}^{(l)} &= (\boldsymbol{\alpha}^{(l)}\boldsymbol{\Phi}^{(l)})\boldsymbol{x}^{(l)} && (26) \\ &= \boldsymbol{\alpha}^{(l)}(\boldsymbol{\Phi}^{(l)}\boldsymbol{x}^{(l)}) && (27) \\ &= \boldsymbol{\alpha}^{(l)}\boldsymbol{x}'^{(l)}. \quad \square && (28) \end{aligned}$$

Hence, we can interpret $\boldsymbol{\alpha}^{(l)}$ as the new weight matrix and $\boldsymbol{x}'^{(l)}$ as the compressed input of layer l.

Compressing the weight matrix of layer $l = 0$ of the ANN is equivalent to compressing its input $\boldsymbol{x}^{(0)}$ (with a bias). Compressing the input can be done using Equation (19). In particular, for single layer networks compressed sensing and model compression are mathematically identical. Therefore, we can transfer the ideas from compressed sensing to model compression.

Usually we use orthogonal cosine functions for compression in this article, but when we deal with a large amount of data like BCI data, we could achieve better results with random compression. Note that random compression has one fundamental prerequisite: all data dimensions have to be nearly equally scaled and should be equally important for the prediction.

The equivalence of input compression and model compression raises the question which method we should prefer. The only criterion to decide this is computational cost. The bottleneck for input compression obviously is compression of new data. If we must compress the same data twice, we will be able to cache the result. For model compression the bottleneck is the adaption of parameters because we then have to generate the weights again. Hence, we have to distinguish SLPs and MLPs during (batch) learning and prediction. When we train a single layer perceptron, we can compress the input because it does not change during training and thus only has to be done once. If the weights of an SLP are optimized and we want to predict a previously unseen instance, it will be faster to generate the weights of the SLP from compressed parameters. That way we can compute the compression of the training data and activation at once. For MLPs this is not feasable because the input of all other layers except the first one will change whenever the weights are adjusted. Thus, model compression is better for MLPs during training as well as during prediction.

5. Supervised Learning

One of the key contributions of this article is the backpropagation algorithm for compressed MLPs. Thus, we want to show the advantages of compression by means of supervised learning. We want to demonstrate that a reduced number of parameters accelerates the optimization process. The main reason for this is the reduced time for each iteration of the optimization algorithm because some of the best optimization algorithms for ANNs have quadratic or cubic time complexity.

In the following experiments we use the Levenberg-Marquardt algorithm (LMA) [37, 39] to optimize the parameters of the ANN. The implementation

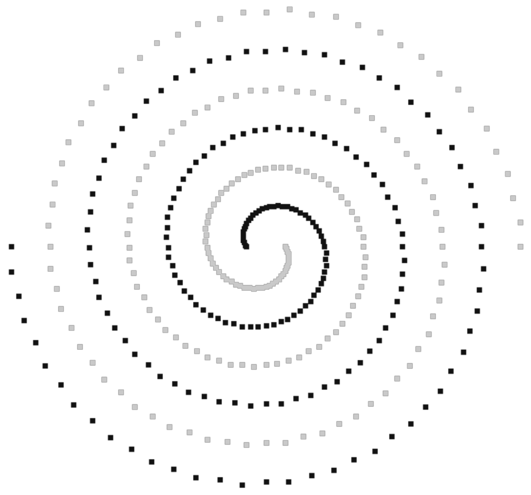

Figure 3: Two interlocked spirals. Different colors indicate different classes.

we use is from the library ALGLIB [9]. This algorithm has cubic time complexity in every iteration. But it usually needs significantly less iterations than other optimization algorithms (for example conjugate gradient).

5.1. Two Spirals

In this experiment we use MLPs to compare the influence of model compression on the training time. The two spirals data set, which has been developed by Lang and Witbrock [34], is a standard benchmark for classification algorithms. The goal is to separate two classes of points that are located on a two-dimensional surface (see Figure 3). They are arranged as interlocked spirals. The data set we use here is taken from the fast artificial neural network library (FANN) [40]. It consists of 193 training instances and 193 test instances.

We use MLPs with two hidden layers to solve this problem. The instances of this data set are not compressible and an indirect weight representation actually makes the optimization more difficult. Here, we only want to demonstrate that the mere reduction of parameters accelerates the optimization.

In this experiment we use MLPs with two hidden layers to compare the influence of compression on the training time. We use hyperbolic tangent in every layer as activation function. The classes are encoded as -1 and 1. We compress the weights with orthogonal cosine functions. We draw the initial

Architecture	*Compression*	*L*	*Correct predictions*	*Iterations*	*Training time*
			$\mu \pm \sigma$	$\mu \pm \sigma$	μ
2-20-10-1	-	281	188.25 ± 1.554	768 ± 21.979	5981 ms
2-20-20-1	-	501	188.57 ± 1.448	464 ± 13.867	12466 ms
2-20-20-1	3-21-21	501	186.06 ± 1.533	305 ± 9.903	8174 ms
2-20-20-1	3-12-12	312	185.66 ± 1.701	511 ± 16.075	5248 ms
2-20-20-1	3-6-6	186	184.75 ± 1.914	679 ± 18.572	3033 ms
2-20-20-1	3-6-3	183	185.14 ± 1.798	775 ± 20.821	3381 ms

Table 2: Results of experiments with the two spirals data set. The values are averaged over 100 runs. The training time is measured with an Intel Core i7-2600K. The notation of architecture is explained in Section 3.2 and the notation of compression is explained in Section 4.1.

parameters from $\mathcal{N}(0, 0.05)$. The error function is the sum of squared errors (SSE). In this experiment we use the following stopping criteria:

- the maximum number of iterations is 1,000,
- the difference of the error between consecutive iterations must not be too small, hence we stop the optimization if

$$|E^{t+1} - E^t| \leq 10^{-8} \cdot \max\{|E^{t+1}|, |E^t|, 1\},$$

- the gradient must not be too small, therefore we stop if

$$|\boldsymbol{g}| \leq 10^{-8},$$

where t is the current iteration, E^t is the error at iteration t, and $\boldsymbol{g}$ is the current gradient.

The results are shown in Table 2. The number of parameters to be optimized is denoted as L. It is either the number of weights for uncompressed ANNs or the number of parameters $\sum_j M_j$ for compressed ANNs. For each configuration we did 100 test runs.

The differences in the classification performance are negligible. As we expected, the optimization problem became more difficult due to indirect weight representation. The indicator for this is the number of iterations. This number even increases when we reduce the number of parameters. However, the training time also decreases because the time for each optimization step decreases significantly. For example, the shortest average training time (1740 ms) is achieved with a compressed MLP that has only 186 parameters but needs 401 iterations on average, that is each iteration takes less than 5 ms. In comparison, the uncompressed MLP with the same topology has an average training time of 7176 ms although only 251 iterations are required on average because each iteration takes more than 28 ms. This shows that the additional complexity of the extended backpropagation is by far not as crucial as the reduced complexity of the optimization step.

5.2. P300 Speller

The two spirals data set is a difficult problem for neural networks. Nevertheless, it is not a complex problem because the input dimension is low and the training set is small. In contrast, brain-computer interfaces (BCIs) [51] generate a huge amount of data. Ordinary systems measure brain activity with 64 or more electrodes and sample signals with a high frequency. The relevant signal usually has a significantly lower frequency. Therefore the data is highly compressible. In the following experiment we use data from a BCI as an example for a complex problem and show that we can use compressed sensing in combination with model compression to speed up the training time and even improve the performance of the classifier.

A brain-computer interface can be used to spell characters. The P300 speller [17, 20] is based on the *oddball paradigm*: whenever a user encounters an important rare stimulus in a sequence of stimuli a P300 potential is elicited. The P300 potential is a so-called event-related potential (ERP) and occurs around 300 ms after the corresponding stimulus.

In this setup the user concentrates on the character he wants to spell at the character matrix that is shown in Figure 4. All columns and rows are intensified 15 times in random order. Each of these repetitions is called a trial. A classifier is used to recognize P300 potentials that occur whenever the row or column with the correct character is intensified. After each character epoch the classifier's score for each row and column will be accumulated and the result will be the character which is contained in the column and the row with the highest scores. A low classifier accuracy does not necessarily lead to

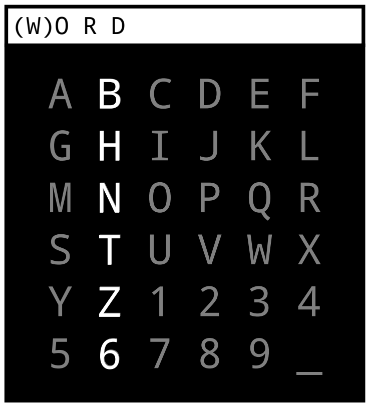

Figure 4: Character matrix.

a low character accuracy because we accumulate 15 trials. Here we use the the BCI competition III data set II [8, 33]. We chose this data set because there are many published results that we can compare to our results.

An instance consists of 15,360 components, that is one second sampled with 240 Hz from 64 channels. We use a single layer perceptron (SLP) with hyperbolic tangent activation function for classification and allow two preprocessing methods:

- Lowpass filter with cut-off frequency 10 Hz.
- Downsampling with a factor of eleven.

The combination of both is called decimation. We chose these preprocessing methods because the winners of the competition had a similar setup [41]. After downsampling, an instance consists of 1,344 components.

The compression we use here is random compression, where each component of $\mathbf{\Phi}$ is drawn from the following distribution:

$$\Phi_{mi} = \begin{cases} +\sqrt{\frac{3}{M_j}} & \text{with probability } \frac{1}{6} \\ 0 & \text{with probability } \frac{2}{3} \\ -\sqrt{\frac{3}{M_j}} & \text{with probability } \frac{1}{6} \end{cases} \tag{29}$$

The result is a matrix with approximately $\frac{2}{3}$ entries of 0, that is not all parameters affect all weights. We have shown that compressing the input of a layer is equivalent to compressing the weights of a layer (see Theorem 4).

Parameters	*Compression*	*Lowpass filter*	*Downsampling*	*Accuracy*		*Training time*	*Iterations*
				15 trials	5 trials		
801	✓	✓	✓	93.9 %	63.8 %	59.6 s	12.4
801	✓	✓	-	94.1 %	64.2 %	85.3 s	16.1
1201	✓	-	-	87.8 %	55.3 %	143.8 s	16.1
1345	-	✓	✓	93.5 %	64.1 %	145.5 s	16.3
15360*	-	-	-	20.0 %	-	25 h	-

Table 3: P300 speller results. The values are averaged over two subjects and ten runs per subject. (*) We only did one run with one subject for this configuration.

In an SLP we only have one layer. Thus compressing the data is equivalent to compressing all weights. There is only one difference: we do not compress the weight of the bias, when we compress the data. Nevertheless we do compress the data here. This can also be regarded as a preprocessing method.

The most successful setups are listed in Table 3. We tried several configurations and noticed that neither higher compression nor lower compressions yield better results. The results are averaged over two subjects and ten runs per subject. The compression matrix $\mathbf{\Phi}$ and the initial parameter vector $\boldsymbol{\alpha}_0$ varied during the ten runs. An uncompressed SLP needed more than one day of training and did not reach an accuracy better than 20 %.

The best results of the BCI competition III are shown in Table 4. Three of our results would have been ranked second in this competition. Note that we neither did use preprocessing methods like channel selection, detrending, PCA, etc. nor did we use an ensemble of classifiers. Almost no domain specific knowledge is required for our method. Only a lowpass filter is really required in order to achieve comparable results. So we had to make a rough assumption about the frequency of the P300 potential. We have no information about the training time used to generate the competition results. Thus, we cannot certainly say that we developed a significantly faster method.

5.3. Singe Trial P300 Detection

A more challenging task in BCI is P300 detection with only a single trial. Here we use data from the project IMMI (Intelligent Man-Machine

Accuracy		Classifier	Preprocessing
15 trials	5 trials		
96,5 %	73,5 %	Ensemble of SVM	Channel selection, bandpass filter, downsampling
90,5 %	55,0 %	Bagging with SVM	Channel selection, bandpass filter, eye movement artifacts removal, downsampling
90 %	59,5 %	-	Lowpass filter, downsampling, detrending, PCA, t-statistic
89,5 %	53,5 %	Gradient Boosting	Detrending, decimation
87,5 %	57,5 %	Bagging with LDA	Channel selection, bandpass filter, downsampling

Table 4: Best results of the BCI Competition III [8].

Interface)[1] of the German Research Center for Artificial Intelligence (DFKI) Robotics Innovation Center and the University of Bremen. We have 24 data sets, that is three independent sessions from eight subjects.

In the experiment two kinds of visual stimuli were presented to the test person: irrelevant “standards” and relevant “targets”. The targets are supposed to elicit P300 potentials. When a target was presented the test person had to react with a movement of the right arm. The ratio between standards and targets was 8:1. The voltage is recorded at 5 kHz sampling rate with 124 electrodes. For the experiments we used 62 EEG electrodes. The data was acquired using an actiCap system (Brain Products GmbH, Munich, Germany) and amplified by four 32 channel BrainAmp DC amplifiers. We preprocessed the data as follows:

1. Standardization: We subtracted the mean of each channel and divided the data by the standard deviation.
2. Decimation to 25 Hz: The data was filtered with a lowpass filter and subsampled.
3. Lowpass filter with cut-off frequency 4 Hz.

After this preprocessing an instance consists of 1,550 components: 25 samples from 62 channels. These preprocessing methods were found to improve the

[1]Website of the project: `http://robotik.dfki-bremen.de/en/research/projects/space-robotics/immi.html`

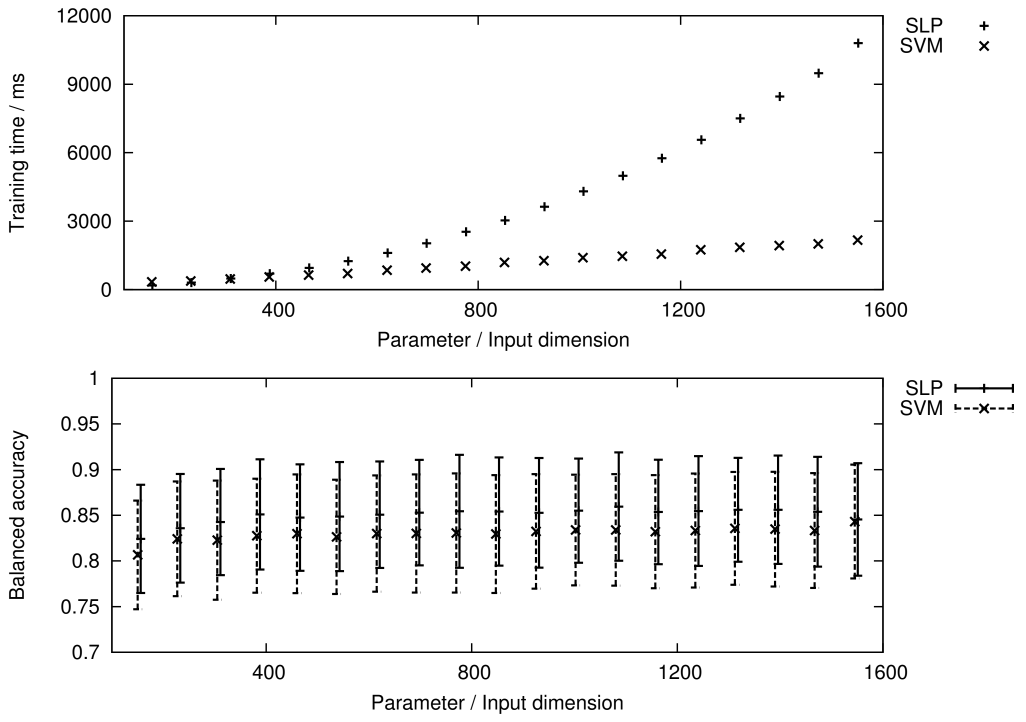

Figure 5: Balanced accuracy and training time of compressed SLP and SVM. The values are averaged over 80 experiments (ten per subject). The interval $[\mu - \sigma, \mu + \sigma]$ is shown for the balanced accuracy.

classification significantly in the project IMMI and the size of an instance vector is still big for normal classification algorithms. Since this is not the focus of this article we will not go into detail here.

Here we compare two types of machine learning algorithms: SLPs and SVMs. We use the library LIBSVM [15] with a linear kernel. We use compressed sensing for both classifiers. We will measure generalization performance and training time. In order to determine the generalization performance, we use the data of the first two sessions as training set and the data of the third session as test set and calculate the *balanced accuracy* [11]

$$\text{balanced accuracy} = \frac{\frac{TP}{TP+FN} + \frac{TN}{TN+FP}}{2}, \tag{30}$$

which is suitable for unbalanced class distributions. Here, TP are true positives, FN are false negatives, TN are true negatives and FP are false positives.

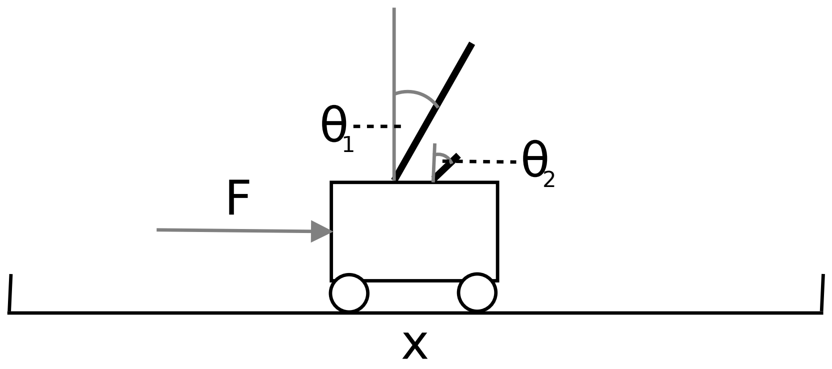

Figure 6: Pole balancing environment.

The results averaged over all subjects are shown in Figure 5. The training time of an SVM is usually much shorter and the reduction of parameters is more effective for SLPs in terms of training time. However, we can approximately preserve the accuracy when we reduce the number of parameters to 15 % of the number of weights (233 parameters) while at the same time the training time is reduced to less than 20 % for SVMs (380 ms) and less than 3 % for SLPs (308 ms). The training time of this compressed SLP is just a little bit shorter than the training time of the compressed SVM but it is significantly shorter than the training time of an uncompressed SVM. The balanced accuracy of SVMs and SLPs is approximately the same, but SLPs seem to be slightly better for compressed data.

6. Reinforcement Learning

An area of application for model compression that has already been explored by Koutník et al. [31] is reinforcement learning. Here the computational complexity of an optimization step is negligible. The only thing that counts is the number of episodes required to learn a successful policy because we usually deal with simulation environments that are computationally expensive or with the real world.

In these experiments we evolve the parameters or weights of MLPs that represent the agent's policy. This is a form of neuroevolution.

6.1. Pole Balancing

Our first reinforcement learning experiment is a very simple benchmark. The pole balancing environment is depicted in Figure 6. The goal is to balance two poles mounted on a cart by applying a force F. The dynamics are described by Wieland [50]. We will analyze single and double pole balancing with and without velocities. This has already been done by Koutník et al. [31] with the optimization algorithm CoSyNE [22].

We perform direct policy search to solve this problem. The optimization algorithm IPOP-CMA-ES [1, 24] is used to adjust the weights of the policy $\pi : S \to A$, where S is the state space and A is the action space. A similar approach has been examined by Heidrich-Meisner and Igel [25]. The fitness function we want to maximize is the number of time steps per episode, that is the weights will be updated after each episode.

The policy π is represented by a single layer perceptron without bias, which is sufficient to solve the problem in the fully observable case. When we do not know the velocities, we have to reconstruct them somehow from the current and the last state. One way to do this is to realize a kind of memory within the neural network. This is possible with recurrent neural networks. The option we chose here is an augmented neural network with Kalman filters (ANKF) [29]. ANKFs are neural networks with a specialized recurrent neural network (α-β filter), which is used to estimate the velocities. We will not compress the parameters of the α-β filters. Thus, we have to optimize L parameters for the single layer perceptron and one parameter for each α-β filter (we will call this number K).

We compare compressed and uncompressed SLPs in terms of the number of episodes required to learn a successful policy. The results are summarized in Table 5. SLPs with less parameters tend to converge faster if they are able to represent a successful policy. So, we only listed the best compression setups. None of the 1,000 experiments for each configuration failed. It was always possible to learn a successful policy with the listed configurations. In three out of four environments it was possible to reduce the number of episodes required to learn a successful policy with model compression. Double pole balancing without velocities is the exception. This result is consistent with the result of Koutník et al. [31]. They even slowed down the learning speed significantly by compressing the weights for double pole balancing without velocities. Another similarity is that single pole balancing can be solved in 2.5 episodes on average. Koutník et al. [31] explained that a solution only requires the weights of an SLP to be equal and positive. When we represent all weights with only one parameter they always have the same value. Therefore, the optimization algorithm only has to find a positive value.

Environment		Setup		Episodes					
Velocities	*Poles*	*Compression*	$L + K$	μ	σ	*min*	*max*	*median*	*Time*
✓	1	-	4	33.1	20.8	1	142	28	119 ms
✓	1	✓	1	**2.5**	2.4	1	40	2	104 ms
✓	2	-	6	261.2	174.3	28	1410	224	210 ms
✓	2	✓	5	**201.4**	229.8	10	1336	139	160 ms
-	1	-	4 + 2	31.4	15.4	1	102	30	117 ms
-	1	✓	3 + 2	**14.3**	9.6	1	57	12	114 ms
-	2	-	6 + 3	**425.5**	220.9	3	1714	388	229 ms
-	2	✓	5 + 3	434.3	318.4	25	1909	352	195 ms

Table 5: Pole balancing results. The values are averaged over 1000 experiments.

Pole balancing with velocities		Episodes	
Method	Publication	Single	Double
CoSyNE	Gomez et al. [22]	98	954
CMA-NeuroES	Heidrich-Meisner and Igel [25]	91	585
CoSyNE and DCT	Koutník et al. [31]	2	258

Pole balancing without velocities		Episodes	
Method	Publication	Single	Double
CMA-NeuroES	Heidrich-Meisner and Igel [25]	192	860
CoSyNE and DCT	Koutník et al. [31]	151	3.421
CoSyNE	Gomez et al. [22]	127	1.249
CMA-ES and ANKF	Kassahun et al. [29]	-	302

Table 6: State of the art results in pole balancing. The results are compared in terms of episodes that were required to learn a successful policy.

An overview of the best results in pole balancing at the moment is shown in Table 6. We could at least reach or even outperform most of the results. Again, the exception is double pole balancing without velocities. Kassahun et al. [29] achieved a significantly better result even though almost the same methods were used. The reason is that they did not learn the parameters of the α-β filters. These parameters were fixed.

(a) (b)

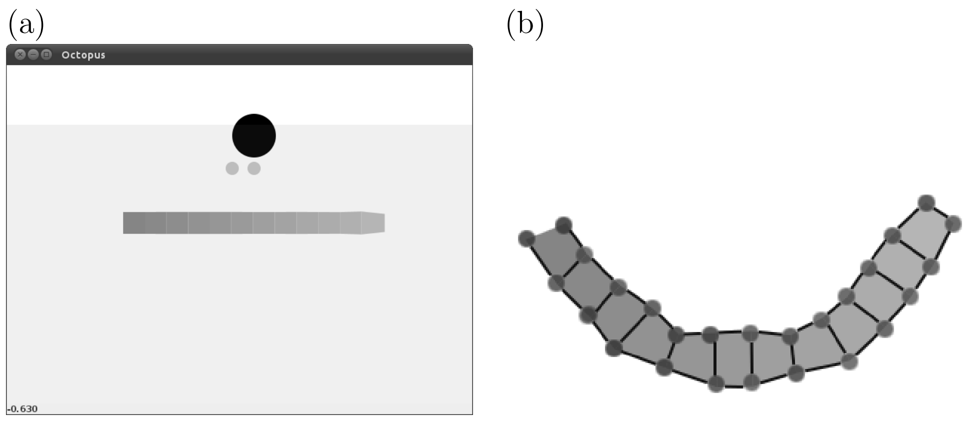

Figure 7: (a) Octopus arm environment. The dark gray line is the straightened arm, the light gray circles are pieces of food and the black circle is a mouth. (b) Octopus arm. The black lines mark muscles and the circles designate point masses of the octopus arm.

Koutník et al. [31] already achieved similar results with a similar approach. The differences are

- they used an optimization algorithm that does not work as good as IPOP-CMA-ES in this case,
- they did use a recurrent neural network instead of α-β filters, which also was disadvantageous.

6.2. Octopus Arm

The octopus arm environment [2] is a complex reinforcement learning problem. The dimensions of the state and action space are greater than those of pole balancing. The environment is depicted in Figure 7. In the environment configuration we use here, the arm consists of 12 compartments and each compartment consists of four point masses that are shared with adjacent compartments and has three muscles. The position and velocity of the point masses are components of the state as well as the positions and velocities of the food. The agent has to control the muscles that can be contracted with a continuous degree of strength. A state consists of 106 components and an action consists of 36 components. The values of the action's components

[2]The octopus arm environment is available at `http://www.cs.mcgill.ca/~dprecup/workshops/ICML06/octopus.html`.

		Return in 900 episodes (average of 20 experiments)		
Compression	L	μ	σ	max
-	1466	-4,52	1,02	11,71
107-11	1466	2,49	0,82	11,73
80-11	1196	1,89	1,32	11,72
40-11	796	2,06	0,86	11,80
20-11	596	2,28	0,60	11,72
10-11	496	2,73	1,30	11,73
5-11	446	**3,60**	1,08	11,72

Table 7: Octopus arm benchmark results. The maximum return is an indicator for the best policy that is representable with these setups and the mean average return indicates how fast a good policy is learned. The notation of compression is explained in Section 4.1.

have to be in $[0, 1]$. For this reason, the activation function of the output layer will be logistic in these experiment.

The return of an episode is the sum of all rewards. The reward for each step is usually -0.01. When the agent feeds the mouth with the left piece of food it receives the reward 5 and when it feeds the right piece of food it receives the reward 7. An episodes stops if either both pieces of food are in the mouth or 1,000 steps have passed. Hence, the minimal return is -10 and the upper bound is 12.

We use almost the same approach to solve this problem as for pole balancing. But the policy is represented by an MLP with ten hidden units and a bias, that is the MLP has 1466 weights. We assume that the task is simple enough to be solved with less parameters. We compress the weights with orthogonal cosine functions.

We calculate the average return in each experiment as an indicator for the speed of the learning algorithm and the maximum return as an indicator for the best representable policy. The results are listed in Table 7. One can conclude from the maximum return that each configuration is in principle able to represent a very good policy. The uncompressed MLP yields by far the worst result. It is possible to learn a good policy but this does not happen fast. When we represent the weights with orthogonal cosine functions and do not reduce the number of parameters, the result is already better. Thus, this representation is advantageous for this problem. But a significant reduction of parameters further increases the mean average return and thus improves the learning speed.

7. Conclusion

We introduced a general framework for compressing the weights of a neural network and extended the backpropagation method. We have shown that compressing the input of a layer of a neural network is the same as compressing the weights of the layer. In particular, compressing the input layer is equivalent to compressing the input to the network. Therefore, we are now able to transfer some ideas from compressed sensing. For example, we can use randomly generated matrices to compress the weights of a neural network, when we know that the data is compressible. In addition, model compression can now be regarged as *implicit* preprocessing.

We have applied model compression to both supervised learning and reinforcement learning. We found that model compression

- reduces the computational cost of an optimization iteration and
- can reduce the number of episodes required to learn good policies in a complex reinforcement learning problem.

We have examined only fully connected layers of neural networks here. However, fully connected layers exist in neural networks like convolutional neural networks or deep belief networks as well. Therefore it is simple to transfer model compression to these types of neural networks. But it is even possible to adapt this method to layers that are not fully connected or share weights, for example convolutional layers in CNNs.

Acknowledgement

This work was supported through a grant of the Federal Ministry of Economics and Technology (BMWi, FKZ 50 RA 1011).

8. References

[1] Auger, A., Hansen, N., 2005. A restart CMA evolution strategy with increasing population size. In: Proceedings of the IEEE Congress on Evolutionary Computation. Vol. 2. IEEE Press, pp. 1769–1776.

[2] Baraniuk, R. G., Davenport, M. A., DeVore, R. A., Wakin, M. B., 2008. A simple proof of the restricted isometry property for random matrices. Constructive Approximation (3), 253–263.

[3] Bartsch, S., Birnschein, T., Cordes, F., Kühn, D., Kampmann, P., Hilljegerdes, J., Planthaber, S., Römmermann, M., Kirchner, F., 2010. SpaceClimber: Development of a six-legged climbing robot for space exploration. In: 41st International Symposium on Robotics (ISR) and 6th German Conference on Robotics (ROBOTIK). VDE Verlag, pp. 1–8.

[4] Bengio, Y., 2007. Learning deep architectures for AI. Tech. Rep. 1312, Dept. IRO, Universite de Montreal.

[5] Bengio, Y., Lecun, Y., 2007. Scaling learning algorithms towards AI. In: Bottou, L., Chapelle, O., Decoste, D., Weston, J. (Eds.), Large-Scale Kernel Machines. MIT Press, pp. 321–360.

[6] Bingham, E., Mannila, H., 2001. Random projection in dimensionality reduction: applications to image and text data. In: Proceedings of the seventh ACM SIGKDD international conference on Knowledge discovery and data mining. ACM, pp. 245–250.

[7] Bishop, C. M., 1996. Neural Networks for Pattern Recognition. Oxford University Press.

[8] Blankertz, B., 2005. BCI competition III webpage. Webpage.
URL `http://www.bbci.de/competition/iii/`

[9] Bochkanov, S., Bystritsky, V., 2011. Levenberg-Marquardt algorithm for multivariate optimization. Webpage.
URL `http://www.alglib.net/optimization/levenbergmarquardt.php`

[10] Boser, B. E., Guyon, I. M., Vapnik, V. N., 1992. A training algorithm for optimal margin classifiers. In: Proceedings of the 5th Annual ACM Workshop on Computational Learning Theory. ACM Press, pp. 144–152.

[11] Brodersen, K. H., Ong, C. S., Stephan, K. E., Buhmann, J. M., 2010. The balanced accuracy and its posterior distribution. In: Proceedings of the 20th International Conference on Pattern Recognition. IEEE Computer Society, pp. 3121–3124.

[12] Calderbank, R., Jafarpour, S., Schapire, R., 2009. Compressed learning: Universal sparse dimensionality reduction and learning in the measurement domain. Tech. rep., Princeton University.
URL `http://dsp.rice.edu/files/cs/cl.pdf`

[13] Candès, E. J., Romberg, J. K., 2005. Practical signal recovery from random projections. In: Proceedings of SPIE Computational Imaging III. SPIE Press, pp. 76–86.

[14] Candès, E. J., Tao, T., 2005. Decoding by linear programming. IEEE Transactions on Information Theory 51 (12), 4203–4215.

[15] Chang, C.-C., Lin, C.-J., 2011. LIBSVM: A library for support vector machines. ACM Transactions on Intelligent Systems and Technology 2 (3), 27:1–27:27.

[16] Ciresan, D. C., Meier, U., Masci, J., Schmidhuber, J., 2011. A committee of neural networks for traffic sign classification. In: International Joint Conference on Neural Networks. IEEE Press, pp. 1918–1921.

[17] Donchin, E., Spencer, K. M., Wijesinghe, R., 2000. The mental prosthesis: Assessing the speed of a p300-based brain-computer interface. IEEE Transactions on Rehabilitation Engeneering 8, 174–179.

[18] Donoho, D. L., 2006. Compressed sensing. IEEE Transactions on Information Theory 52 (4), 1289–1306.

[19] Eich, M., Grimminger, F., Kirchner, F., 2008. A versatile stair-climbing robot for search and rescue applications. In: IEEE International Workshop on Safety, Security and Rescue Robotics, 2008. SSRR 2008. pp. 35–40.

[20] Farwell, L. A., Donchin, E., 1988. Talking off the top of your head: Toward a mental prosthesis utilizing event-related brain potentials. Electroencephalography and Clinical Neurophysiology 70, 510–523.

[21] Gionis, A., Indyk, P., Motwani, R., 1999. Similarity search in high dimensions via hashing. In: Proceedings of the 25th International Conference on Very Large Data Bases. Morgan Kaufmann, pp. 518–529.

[22] Gomez, F., Schmidhuber, J., Miikkulainen, R., 2008. Accelerated neural evolution through cooperatively coevolved synapses. Journal of Machine Learning Research 9, 937–965.

[23] Guyon, I., Elisseeff, A., 2003. An introduction to variable and feature selection. Journal of Machine Learning Research 3, 1157–1182.

[24] Hansen, N., Ostermeier, A., 2001. Completely derandomized self-adaptation in evolution strategies. Evolutionary Computation 9 (2), 159–195.

[25] Heidrich-Meisner, V., Igel, C., 2009. Neuroevolution strategies for episodic reinforcement learning. Journal of Algorithms 64 (4), 152–168.

[26] Hornik, K., Stinchcombe, M. B., White, H., 1989. Multilayer feedforward networks are universal approximators. Neural Networks 2 (5), 359–366.

[27] Indyk, P., Motwani, R., 1998. Approximate nearest neighbors: towards removing the curse of dimensionality. In: Proceedings of the 30th annual ACM symposium on Theory of computing. ACM, pp. 604–613.

[28] Johnson, W. B., Lindenstrauss, J., 1984. Extensions of lipschitz mapping into hilbert space. In: Conference in Modern Analysis and Probability. Vol. 26 of Contemporary Mathematics. American Mathematical Society, pp. 189–206.

[29] Kassahun, Y., de Gea, J., Edgington, M., Metzen, J. H., Kirchner, F., 2008. Accelerating neuroevolutionary methods using a Kalman filter. In: GECCO. ACM, pp. 1397–1404.

[30] Kassahun, Y., Wöhrle, H., Fabisch, A., Tabie, M., 2012. Learning parameters of linear models in compressed parameter space. In: Villa, A., Duch, W., Érdi, P., Masulli, F., Palm, G. (Eds.), Artificial Neural Networks and Machine Learning – ICANN 2012. Vol. 7553 of Lecture Notes in Computer Science. Springer, pp. 108–115.

[31] Koutník, J., Gomez, F. J., Schmidhuber, J., 2010. Evolving neural networks in compressed weight space. In: Pelikan, M., Branke, J. (Eds.), GECCO. ACM, pp. 619–626.

[32] Koutník, J., Gomez, F. J., Schmidhuber, J., 2010. Searching for minimal neural networks in fourier space. In: Baum, E., Hutter, M., Kitzelnmann, E. (Eds.), Proceedings of The Third Conference on Artificial General Intelligence (AGI 2010). Atlantic Press, pp. 61–66.

[33] Krusienski, D., Schalk, G., 2004. Wadsworth BCI dataset (P300 evoked potentials). Data set description.
URL `http://www.bbci.de/competition/iii/desc_II.pdf`

[34] Lang, K. J., Witbrock, M. J., 1988. Learning to tell two spirals apart. In: Touretzky, D., Hinton, G., Sejnowski, T. (Eds.), Proceedings of the 1988 Connectionist Models Summer School. Morgan Kaufmann, pp. 52–61.

[35] LeCun, Y., Bengio, Y., 1995. Convolutional networks for images, speech and time series. In: Arbib, M. A. (Ed.), The Handbook of Brain Theory and Neural Networks. MIT Press, pp. 255–258.

[36] LeCun, Y., Bottou, L., Orr, G. B., Müller, K.-R., 1998. Efficient backprop. In: Orr, G. B., Müller, K.-R. (Eds.), Neural Networks: Tricks of the Trade. Springer, pp. 9–50.

[37] Levenberg, K., 1944. A method for the solution of certain problems in least squares. Quarterly of Applied Mathematics 2, 164–168.

[38] Maillard, O.-A., Munos, R., 2009. Compressed least-squares regression. In: Bengio, Y., Schuurmans, D., Lafferty, J., Williams, C. K. I., Culotta, A. (Eds.), Advances in Neural Information Processing Systems 22. pp. 1213–1221.

[39] Marquardt, D., 1963. An algorithm for least-squares estimation of nonlinear parameters. Journal of the Society for Industrial and Applied Mathematics 11 (2), 431–441.

[40] Nissen, S., 2003. Implementation of a fast artificial neural network library (fann). Tech. rep., Department of Computer Science University of Copenhagen (DIKU).
URL `http://leenissen.dk/fann/report/report.html`

[41] Rakotomamonjy, A., Guigue, V., 2008. Bci competition iii: Dataset ii - ensemble of svms for bci p300 speller. IEEE Transactions on Biomedical Engeneering 55 (3), 1147–1154.

[42] Römmerman, M., Kühn, D., Kirchner, F., 2009. Robot design for space missions using evolutionary computation. In: Evolutionary Computation, 2009. CEC '09. IEEE Congress on. pp. 2098–2105.

[43] Rumelhart, D. E., Hinton, G. E., Williams, R. J., 1986. Learning representations by back-propagating errors. Nature 323 (6088), 533–536.

[44] Sarle, W. S., 1997. Neural network FAQ. Postings to the Usenet newsgroup comp.ai.neural-nets.
URL `ftp://ftp.sas.com/pub/neural/FAQ.html`

[45] Schmidhuber, J., 1995. Discovering solutions with low Kolmogorov complexity and high generalization capability. In: International Conference on Machine Learning. Morgan Kaufmann, pp. 488–496.

[46] Schmidhuber, J., 1997. Discovering neural nets with low Kolmogorov complexity and high generalization capability. Neural Networks 10 (5), 857–873.

[47] Spenneberg, D., Kirchner, F., 2007. The bio-inspired SCORPION robot: Design, control & lessons learned. In: Zhang, H. (Ed.), Climbing & Walking Robots: towards New Applications. InTech, pp. 197–218.

[48] Sutton, R. S., Barto, A. G., 1998. Reinforcement Learning: An Introduction. MIT Press.

[49] Vapnik, V. N., 1995. The Nature of Statistical Learning Theory. Springer-Verlag New York, Inc.

[50] Wieland, A. P., 1990. Evolving controls for unstable systems. In: Touretzky, D. S., Elman, J. L., Sejnowski, T. J., Hinton, G. E. (Eds.), Connectionist Models: Proceedings of the 1990 Summer School. CA: Morgan Kaufmann, pp. 91–102.

[51] Wolpaw, J. R., Birbaumer, N., McFarland, D. J., Pfurtscheller, G., Vaughan, T. M., 2002. Brain-computer interfaces for communication and control. Clinical Neurophysiology 113 (6), 767–91.