Bachelor's Thesis

Differentiable Convolutional Neural Network Architectures for Time Series Classification

Differenzierbare Convolutional Neural Network Architekturen für Zeitreihenklassifikation

Louis Kirsch

louis.kirsch@student.hpi.de

Submitted on July 20, 2017

Knowledge Discovery and Data Mining Group
Hasso Plattner Institute, Germany

Supervisors

Arvind Kumar Shekar

Prof. Dr. Emmanuel Müller

Abstract

In the last few years, deep learning revolutionized many areas of machine learning. One very important area is time series classification. Time series are produced everywhere in real world scenarios, such as industrial or medical sensor data. The success in deep learning is largely based on training through backpropagation. Nevertheless, backpropagation can only be applied to weights training, since architecture construction is not differentiable. As a result, state of the art architectures are commonly handcrafted. This thesis addresses the problem of automatically designing architectures for time series classification in an efficient manner. Existing solutions for constructing architectures algorithmically, such as evolutionary or reinforcement learning methods, are much more computationally expensive. We address the problem by introducing a regularization technique for convolutional neural networks (CNNs) that enables joint training of network weights and architecture through backpropagation. Skip connections and a special two-phased training are introduced to enable a stable optimization. We evaluate the approach on the UCR archive, yielding competitive results compared to state of the art in time series classification, and outperforming on datasets where handcrafted architectures do not match the complexity of the dataset.

Contents

1 Introduction

Supervised deep learning achieved super-human performance on many tasks in recent years. Recurrent neural networks, specifically LSTMs [1], revolutionized sequence to sequence modeling. Deep convolutional neural networks (CNNs) introduced by LeCun [2] lead to better image classification, segmentation and signal processing. Nevertheless, state of the art architectures are commonly handcrafted [3,4,5,6], namely choosing the type of layers to use, the number of neurons, the depth of the network and possible preprocessing. In the case of CNNs, more parameters emerge: This includes the number of convolutional layers and the stride, kernel size, and number of output channels for each of them. Max or average pooling can be used and the number of fully connected layers and their number of neurons need to be chosen.

Often those architectures and hyperparameters are chosen manually, based on the researcher's experience and intuition. As an alternative in some cases, grid search is applied, a very simple form of determining the right hyperparameters and architecture. This search is not guided by heuristics and is therefore very expensive.

Determining the weights of a neural network is an optimization problem that is, in the meanwhile, almost always solved by applying gradient descent in the form of backpropagation. But we rarely consider that determining neural network architectures and other hyperparameters are just an instance of optimization as well. Architectures are not differentiable, thus very recently Real et al. [7] applied evolutionary methods and Zop et al. [8] proposed an approach based on reinforcement learning, producing new state of the art architectures. Typical for evolutionary methods the former approach requires many trial and error mutations with no guidance except a population of already well-working architectures. Therefore, this method needs to scale to many hundred computing instances that mutate and train an existing population member. The latter approach needs to train an entire recurrent neural network that learns to output the architecture of another network, requiring large computational resources as well. Based on the idea that architectures can be designed to be differentiable to some extent, Miconi [9] showed that the number of neurons within a layer can be controlled using a specifically designed regularizer. While it was demonstrated that the network complexity can be controlled in such way, the approach was limited to recurrent neural networks and was only run on a self-designed simple sequence prediction problem. It was not tested on any commonly used benchmark or compared to state of the art handcrafted architectures. Also, the depth of the network, a very significant hyperparameter, was not optimized.

For the application of time series classification (TSC), we propose the differentiable

convolutional neural network architectures (DiffCNN) that eliminate the necessity of handcrafting architectures. In contrast to state of the art, the DiffCNN is purely based on differentiation. We make the following contributions:

- Make all architecture parameters for TSC differentiable that are hard to manually specify.
- Require only slightly more training time than conventional CNN training with fixed network architectures.
- Achieve competitive results on the UCR archive [10] benchmark.
- Outperform on datasets where a handcrafted architecture, designed for a range of datasets, does not match the inherent complexity of the individual dataset.

TSC is the task of assigning a time series one category of a fixed set. We define a set of time series S containing N time series s_i for $i \in [\ 1\ ..\ N\ ]$. Each s_i is a real valued vector $s_i \in \mathbb{R}^l$ with length l. We assign every s_i a class label $y_i \in C$ where C is the set of all classes in the dataset. We partition S into a training set $T \subset S$ and a test set $E \subset S$ of sizes N_T and N_E respectively. The task of time series classification is to fit a model M on T that can assign each time series $s_i \in T$ a class label $M(s_i) = \hat{y}_i$ so that $\hat{y}_i = y_i$ for maximally many i. This classifier is then evaluated on the test set E to determine its generalization performance.

The thesis is structured as described in the following. Section 2 presents related work both in the field of TSC, as well as neural network architecture optimization. In the first part of section 3 we shortly introduce fully connected and convolutional neural networks. Then, in the second part, we define how Miconi [9] regularized recurrent neural networks to make the number of neurons per layer differentiable. In the third part we present the current state of the art in TSC. Based on that, in section 4, we introduce our concept of differentiable CNN architectures and how those are applied to TSC. We present several required training modifications to enable a stable optimization of the DiffCNN. Section 5 then deals with the experiments on several datasets, the advantages and limitations of the approach, and experimental justifications for the proposed algorithm design. Additionally, results are discussed and analyzed. Finally, in section 6, the thesis is summarized and possible future extensions are proposed.

2 Related work

Both TSC, as well as automatically designing neural network architectures have been studied separately before. First, we will identify related solutions to TSC. Then, existing solutions to architecture optimization are presented.

2.1 Time series classification

Time series classification is an extensively studied field with many different approaches to the problem. The field can be partitioned into 4 areas: Distance, feature, ensemble and deep learning based solutions.

Distance based algorithms represent the simplest form of algorithms, directly operating on the time series. For instance, time series can be classified by euclidean distances to the nearest neighbors. An efficient approach to k-nn classification of time series is the use of dynamic time warping (DTW) [11]. Feature based methods, on the other hand, extract numerous features that are then used in conjunction with a classifier. Among the feature based algorithms is TSBF [12]. It randomly selects subsequences of the time series to classify and extract features thereof. Later all features are assembled to a bag of features that are fed to a random forest classifier that makes the final prediction. The Bag-Of-SFA-Symbols (BOSS) classifier [13] extracts words using a discrete fourier transform (DFT) and forms a distribution of words after discretization using multiple coefficient binning (MCB). A distance measure is defined and nearest neighbor classification is performed. The BOSSVS [14] adapts BOSS with a vector space model based on tf-idf in order to reduce the computational complexity. Ensemble based methods include PROP [15] that is an ensemble of elastic distance measures. The COTE [16] ensemble includes 35 classifiers from the time, frequency, change and shapelet domains. Recently deep learning techniques were applied to TSC in the form of CNNs, outperforming previous methods on the majority of UCR datasets [10]. The MCNN [5] applies a wide range of transformations as a preprocessing step to classify datasets where little data is available. This includes scaling, smoothing and sub-sampling. The FCN [6] demonstrated that a well-handcrafted architecture that relies on global average pooling instead of plenty preprocessing can outperform the MCNN. In this thesis, we will introduce a new deep learning approach that replaces handcrafted architectures with architectures learned by differentiation while being competitive to state of the art TSC.

2.2 Learning the architecture of neural networks

The idea of pruning neurons in an existing neural network dates back to 'Optimal Brain Damage' [17]. It is based on calculating a weight saliency measure using second derivatives but does not include adding new neurons or layers. Recently the problem of pruning neurons to reduce network complexity was scaled to today's larger deep learning problems [18]. But the problem of designing the initial architecture was once more not addressed. Biologically inspired algorithms include the design of neural networks by genetic algorithms. Kitano [19] proposes to use a graph grammatical encoding to design neural networks. Often the weights are trained using backpropagation and the architecture is adapted using evolutionary methods [20]. In 2017 two new papers explored evolutionary [7] and reinforcement learning [8] methods. While these methods generated entirely new architectures that redefined the state of the art, they require large computational resources. Additionally, using two methodologies, such as backpropagation for weights and evolution for the architecture, results in the development of a dual representation scheme. One possible alternative is training and constructing the entire network using evolutionary methods [21]. This has the downside of losing all heuristic benefits of backpropagation. The other alternative is using backpropagation for both weights and architecture. Miconi [9] introduced a concept based on L1-regularization that allows both adding new neurons, as well as pruning them solely using backpropagation. He applied the idea to simple recurrent neural networks, but the number of layers is still fixed.

In this thesis, we omit evolutionary methods entirely, adapting both the architecture and the weights using backpropagation while still being able to add and remove layers from the network. This allows leveraging the recent success of deep learning with backpropagation and modifying the architecture in the same methodology while not relying on time and computational resource consuming evolutionary approaches. The properties of the different approaches can be found in Table 1.

Table 1: A comparison of different methods for architecture manipulation

					By backpropagation	
Algorithm	Pruning	Growing	Depth	TSC	Training	Architecture
[17] [18]	✓				✓	✓
[9]	✓	✓			✓	✓
[7] [8]	✓	✓	✓	(✓)[1]	✓	
[19] [20]	✓	✓	✓		✓	
[21]	✓	✓	✓			
This work	✓	✓	✓	✓	✓	✓

[1] It was applied to CNNs but not TSC and thus could be easily transferred

3 Background

This section shortly introduces concepts from Deep Learning, Miconi's previous work on a differentiable number of neurons and TSC.

3.1 Fully connected neural networks and convolutional neural networks

The deep learning approach to TSC is to derive a composite function f_{nn} that maps the input $x^{(1)}$ to a valid probability distribution over the classes C by stacking several layers. Stacking refers to the process of applying several layers on top of each other, generating the neural network. In a neural network with depth d, the ith layer's input for $i = 1, \ldots, d$ is defined as $x^{(i)}$. Each layer is defined by its function $f^{(i)}$. The function f_{nn} is defined as the composed function of all layer functions $f^{(i)}$ of a neural network. In this thesis, we are going to rely on two different kinds of layers that constitute our generated architectures for TSC.

The first type is the fully connected layer: It is usually used when the input has no time component but I different dimensions. Let $x^{(i)} \in \mathbb{R}^{B \times I}$ be the layer's input where B is the batch size and I is the number of input neurons. A batch is the concatenation of multiple samples that are transformed by the network in a single calculation pass. We multiply this input with a weight matrix $w^{(i)} \in \mathbb{R}^{I \times O}$ where O defines the desired number of output neurons. Additionally, we add a bias $b^{(i)} \in \mathbb{R}^{O}$ and apply an activation function σ. The fully connected layer is therefore given by

$$f_{fc}(x^{(i)}) = \sigma(x^{(i)} \times w^{(i)} + b^{(i)}) \tag{1}$$

The operation yields the matrix $x^{(i+1)} \in \mathbb{R}^{B \times O}$, which is also called the activations of layer i. Intuitively, every output unit is a linear combination of all input units. The variables I and O are defined for each layer i, for simplicity we omit the index in $I^{(i)}$ and $O^{(i)}$.

The second type is the convolutional layer: It applies a convolution with a flipped kernel, also known as cross-correlation, on a discrete input signal $x^{(i)}$. This signal includes a time component, therefore any time series $s \in S$ can be fed into a convolutional layer. If the input time series is univariate we have exactly $I = 1$ input channels while multivariate time series can be mapped to multiple channels $I > 1$. On the signal $x^{(i)} \in \mathbb{R}^{B \times L \times I}$ with L being the length of the signal and I the number of input channels, the convolutional layer is defined as

$$f_{conv}(x^{(i)})_t = \sigma\Big(\sum_{k=1}^{K} x^{(i)}_{:,k+t-1} w^{(i)}_k\Big) \quad \text{for} \quad t = 1, \ldots, L \tag{2}$$

where $w^{(i)} \in \mathbb{R}^{K \times I \times O}$ is the kernel with size K applied on $x^{(i)}$ with I input channels, yielding $x^{(i+1)} \in \mathbb{R}^{B \times L \times O}$ with O output channels. The output signal's number of channels is determined by the kernel that is applied. Both the input and output signal have the same length L by padding the input signal accordingly. This is called 'same padding' and is used throughout this thesis. The variables K, L, I and O are specific to every layer, the index (i) is omitted.

Most machine learning problems require a non-linear transformation to the input $x^{(1)}$ for a correct class mapping. Since matrix multiplication and convolution only apply a linear transformation, we include the activation function σ. This so called non-linearity is used in order to approximate non-linear functions with the composite function f_{nn}. In the scope of this thesis we rely on the commonly used rectified linear unit (ReLU) [22] defined as

$$\sigma_{ReLU}(z) = \max(0, z) \tag{3}$$

The output of the top-most, the dth layer, forms a probability distribution over the different classes. In order to enforce a valid distribution we utilize the softmax function instead of the ReLU. The softmax function over classes C is given by

$$\sigma_{softmax}(z)_k = \frac{e^{z_k}}{\sum_{c=1}^{C} e^{z_c}} \quad \text{for} \quad k = 1, \ldots, C \tag{4}$$

Figure 1 gives an example of a convolutional architecture of depth four. Given the input, we first apply a convolution, then a ReLU, then a second convolution, another ReLU and finally a fully connected layer followed by a softmax function.

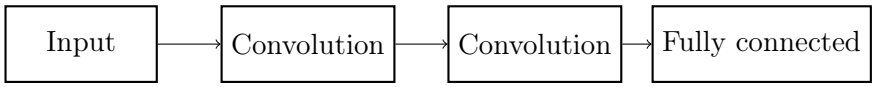


Figure 1: An example neural network architecture for TSC of depth four

We furthermore define a loss function that represents the training objective. In the case of TSC we employ the cross-entropy given by

$$H(p, q) = -\sum_{c \in C} p(c) \log q(c) \tag{5}$$

where p and q are the true discrete probability distribution over classes C and the predicted distribution respectively. The predicted distribution q is generated by the composite function f_{nn}. The entire loss function over a batch of B samples is then defined as

$$l = \frac{1}{B} \sum_{b=1}^{B} H(p_b, q_b) \tag{6}$$

This loss is finally differentiated using backpropagation [23] and gradient descent is applied to minimize the loss. A complete introduction to CNNs and deep learning is given by Goodfellow et al. [24].

3.2 Differentiable number of neurons in recurrent neural networks

Miconi proposed a method [9] to make the complexity of a recurrent neural network differentiable by modifying the number of output neurons per layer during gradient descent. He showed that more complex problems require a bigger number of output neurons, while simpler problems reduce the number of neurons to a minimum. Neurons can be switched on or off by changing its outgoing weights. The magnitude of the outgoing weights $\omega_j^{(i)}$ of the jth neuron in layer i is defined by

$$\omega_j^{(i)} = \sum_k |w_{j,k}^{(i+1)}| \tag{7}$$

That means the following layer's weights define the outgoing weights of the neurons. Figure 2 visualizes the outgoing weights of a neuron in layer 2. If $\omega_j = 0$ for any neuron j, then the neuron's output for the following layer is guaranteed to be zero and the neuron could, therefore, be removed from the network. In that sense, ω_j describes the relevance of the neuron.

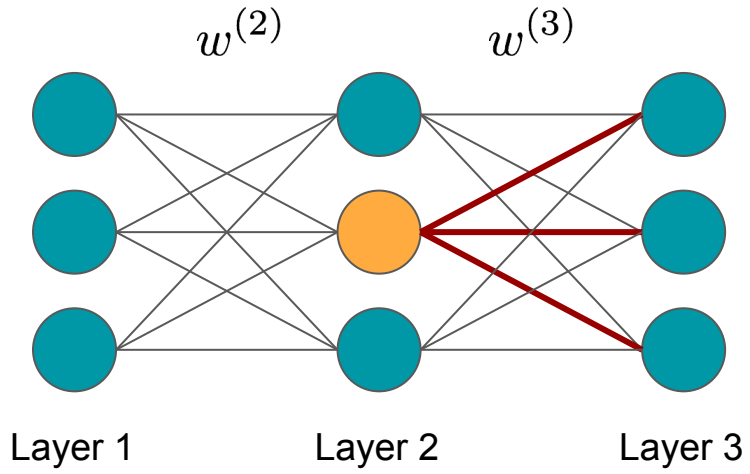


Figure 2: Two fully connected layers following an input layer with a neuron in layer 2 highlighted in orange. The outgoing weights are indicated in red, and defined by the weights $w^{(3)}$ of layer 3.

In order to add or remove neurons, he introduced a regularizer to the loss function that penalizes the magnitude of outgoing weights. The magnitude of outgoing weights is also known as the L1 norm. In order to update the definition of the loss function for TSC from Equation 6, we now introduce a penalty given by

$$P(w) = \sum_i \sum_j |\omega_j^{(i)}| \tag{8}$$

with $w^{(i)}$ being the weights employed in layer i. Thereby, the new, regularized loss function is updated to

$$l_{reg} = \alpha P(w) + \frac{1}{B} \sum_{b=1}^{B} H(p_b, q_b) \tag{9}$$

The new hyperparameter α balances the two training objectives of minimizing the cross-entropy, while also minimizing the magnitude of outgoing weights of the neurons. It was shown that L1 regularization leads to sparsity [25], thereby switching some neurons permanently off through small magnitudes of outgoing weights. This effectively leads to a network architecture that solves the task with as little switched-on neurons as possible.

While the new loss function allows switching off neurons by setting its outgoing weights close to zero, no new neurons can be added so far. To solve this, a variable $t_{inactive} = 1$ and a threshold $t_{del} = 0.05$ are used that govern the pruning and growing of neurons. Let $n_{inactive}$ be the number of inactive neurons in a layer i. Whenever less than $t_{inactive}$ neurons are inactive in that layer, a new neuron is added. Likewise, whenever there are more than $t_{inactive}$ neurons inactive, $n_{inactive} - t_{inactive}$ of those are pruned away. Due to the approximate nature of gradient descent, the magnitude of the outgoing weights of a neuron may never reach exactly zero. Thus, we consider outgoing weights $\omega_j <= t_{del}$ to represent switched off neurons.

Putting together the regularized loss function and thresholding, we yield a framework that can change the number of neurons in each layer only by differentiating and optimizing using gradient descent.

3.3 FCN: The current state of the art in time series classification

While state of the art in TSC was previously dominated by feature extraction methods, the current state of the art approaches include mostly deep learning methods. The state of the art method on the UCR datasets [10], one of the most prevalent repositories for time series classification, is the fully convolutional network (FCN) as introduced by Wang et al. [6]. It features a very simple, but fixed architecture as seen in Figure 3. Except for the final softmax layer, the architecture is fully convolutional and therefore less prone to overfitting. Instead of using a fully connected layer before the softmax layer it employs global average pooling [26]. This means reducing the time axis to its mean before feeding the activations to the softmax layer, giving the added benefit of being able to use arbitrary-length time series as input.

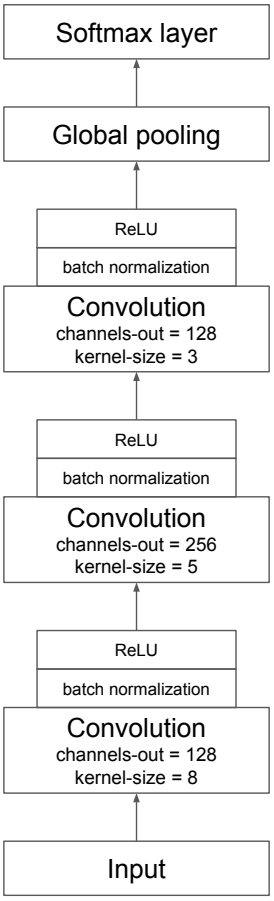


Figure 3: The FCN architecture that currently outperforms all other classifiers on the UCR archive

Batch normalization [27] is applied as a reparameterization technique to improve convergence during training, dealing with the problem of training deep neural networks. During gradient descent, each parameter is updated under the assumption that all other parameters are kept constant. This simplification is problematic when weight updates increase or decrease the variance σ^2 or the mean μ of activations significantly. To prevent this, activations a of a layer are normalized per batch before the non-linearity is applied:

$$\hat{a} = \frac{a - \mu}{\sigma} \tag{10}$$

The variables μ and σ are defined per batch of size B during training time as given in the following

$$\mu = \frac{1}{B} \sum_{i=1}^{B} a_{i,:} \tag{11}$$

$$\sigma = \sqrt{\frac{1}{B}\sum_{i=1}^{B}(a-\mu)_{i,:}^2} \tag{12}$$

During test time, running averages of μ and σ are used. This normalization reduces the expressive power of the neural network. That means fewer functions can be approximated with the same network after batch normalization is introduced. To counteract this, we can reparameterize the normalized activations $\hat{a}$, while keeping the benefits described above. The reparameterization is given by

$$\tilde{a} = \gamma\hat{a} + \beta \tag{13}$$

where γ scales the variance of the activation and β changes the mean. Note that when calculating the activations a we can omit the bias b, as it becomes redundant with the new parameter β. When we refer to the bias in the context of batch normalization, we refer to β.

4 Differentiable CNN architectures

In this section, we propose to extend the work of Miconi [9] for TSC. Instead of controlling network complexity by the number of neurons in a layer, the number of output channels will be influenced by gradient descent. We propose the DiffCNN. A CNN for TSC that can learn its own architecture using gradient descent jointly with weights training. The DiffCNN has the novel ability to change the depth of its architecture, allowing to construct an entire network from scratch. We leverage insights from Wang [6] to create a framework for TSC that can be readily applied to many TSC problems with little hyperparameter tuning. Furthermore, we present several training adaptations for the introduced differentiable CNN architectures that are required for a stable optimization. Important introduced hyperparameters are listed in the Appendix A in Table 2.

4.1 Initial architecture

In order to be suitable for TSC, we define an initial architecture as seen in Figure 4. It represents the minimally viable solution and can then be grown and later pruned as necessary. Its basic structure is derived from [6]: A single convolution with a fixed number of initial output channels $O_{initial}$ is average pooled globally and the resulting activations are combined linearly in the softmax layer to then model a final probability distribution over the classes. Additionally, we use batch normalization and the ReLU as an activation function.

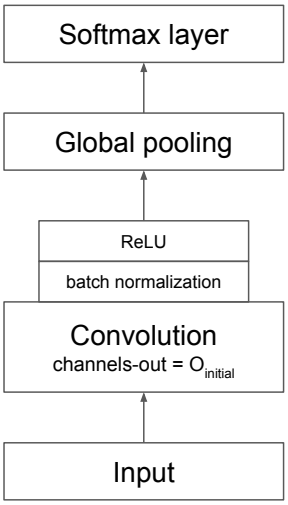


Figure 4: The initial architecture as a minimally viable solution for TSC

Similar to [6] we z-normalize the input time series using the mean and variance of the training set and do not apply any further preprocessing. Due to the univariate nature of the time series we defined in section 1, the number of output channels of the input layer with $i = 1$ is $O^{(1)} = 1$, while the number of input channels of the following first convolution with $i = 2$ is therefore also $I^{(2)} = 1$.

4.2 Replacing neurons with channels

First, we aim to simplify the activity of a neuron by introducing a small adjustment to Miconi's original definition from Equation 8. Instead of defining the activity of a neuron of layer i by its outgoing weights, we use the incoming weights of the same layer as shown in Figure 5. This has the benefit of being able to define whether an output neuron of a specific layer is active, just by referring to this layer's weights instead of taking into account all layers that consume the activations. Therefore, the sum of the neuron's incoming weights, and subsequently its activity, is given by

$$\hat{\omega}_j^{(i)} = \sum_k |w_{k,j}^{(i)}| \tag{14}$$

This change does not affect the optimization objective in any way. We will illustrate this fact with the following example of deactivating a neuron. The highlighted neuron in Figure 5 is only reachable by its incoming weights (left) and outgoing weights (right). Either we set all incoming weights to zero, thus deactivating all information flowing to the neuron, or we set all outgoing weights to zero, thus deactivating all information leaving the neuron. In both cases, the neuron is only effective if both incoming and outgoing weights deviate from zero.

Also, instead of using fully connected layers that are used recurrently we aim to change the complexity of convolutional layers. Convolutional layers are specified by their number of output channels instead of number of output neurons. As given previously, the weights of a convolutional layer are defined as $w \in \mathbb{R}^{K \times I \times O}$ with K being the kernel size, I being the number of input channels and O being the number of output channels. Similarly to controlling the number of output neurons in a fully connected layer, we only need to adapt O to change the expressive power of the layer. Accordingly, we define the activity of an entire output channel j by

$$\tilde{\omega}_j^{(i)} = \sum_k \sum_l |w_{k,l,j}^{(i)}| \tag{15}$$

The new penalty function is therefore defined as

$$\tilde{P}(w) = \sum_i \sum_j |\tilde{\omega}_j^{(i)}| \tag{16}$$

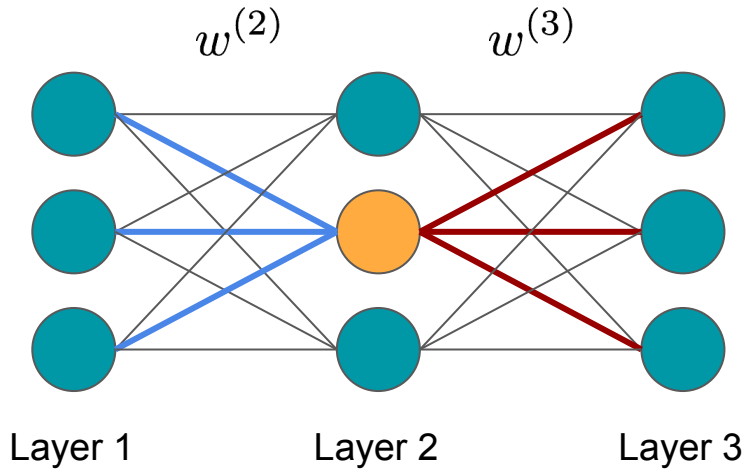


Figure 5: The same fully connected layers as presented previously with the incoming weights of a neuron highlighted in blue

4.3 Independent switches and the penalty function

While the L1-norm of the incoming weights of a channel in a convolutional layer can be used as a measure of relevance of this specific channel, the weights itself are also subject to a training objective that tries to convolve the input to detect certain patterns. During gradient descent, therefore we change both the relevance of the channel and the transformation it applies with the kernel weights. Instead, we would prefer to have a single scalar that defines the relevance of the entire channel that can be optimized independently of the kernel weights. We propose to introduce such a switch variable. With the introduction of batch normalization we have already obtained a suitable reparameterization. In Equation 13 we scaled the entire normalized activations $\hat{a}$ by γ. In the case of a convolution we can define $\gamma \in \mathbb{R}^O$ as a vector that scales each output channel of a layer with a scalar. We define such a $\gamma^{(i)}$ for every convolutional layer i, but omit the layer index for notational simplicity. Whenever γ_j of an output channel j is close to zero, the activations of the entire channel is close to zero for all inputs. Therefore, γ_j acts as a switch that controls the relevance of an output channel j.

The next obvious step would be to replace our penalty function of Equation 16 with

$$\tilde{P}(\gamma) = \sum_i \sum_j |\gamma_j^{(i)}| \tag{17}$$

While this would work, this means that the output channels are linearly more penalized with their scale γ. Instead, we would like to have a measure that penalizes the number of active channels, independent of the scale. Therefore we introduce a

new penalty function loosely based on [28]

$$P_{weigend}(\gamma) = \sum_i \Big[i \sum_j \frac{\left(\gamma_j^{(i)}\right)^2}{\kappa + \left(\gamma_j^{(i)}\right)^2} \Big] \tag{18}$$

This ensures that scales γ_j close to zero are assigned a small penalty while growing scales converge to a penalty of one. Additionally, we introduced the factor i, the layer index, that scales the penalty linearly with the depth of the layer. This expresses the prior that we favor fewer layers with more channels over very deep networks that are more expensive to train while not yielding any advantage in reducing the cross-entropy. Equation 18 introduced a new parameter κ that defines the exact form of the penalty function. We set this parameter by introducing another constraint to the function. When the magnitude of a scale is exactly $|\gamma_j| = 0.5$, half of its initial scale of $|\gamma_j| = 1$, we would like to impose a penalty exactly between the theoretical minimum of $penalty = 0$ and the limiting maximum of $penalty = 1$. This ensures two aspects that can be seen in Figure 6. Firstly, between scale magnitudes from zero to one we obtain similar penalties to the original linear penalty function. Secondly, the gradient of the function at its initial scale is big enough to outweigh the cross-entropy gradient in the case of an unnecessary channel. Therefore we yield

$$\frac{0.5^2}{\kappa + 0.5^2} = 0.5 \tag{19}$$

$$\iff \kappa = 0.5^2 \tag{20}$$

The new loss function we use in this approach is now defined as

$$l_{diffcnn} = \alpha P_{weigend}(\gamma) + \frac{1}{B} \sum_{b=1}^{B} H(p_b, q_b) \tag{21}$$

For notational simplicity we sometimes omit function parameters: The cross-entropy H and accordingly the loss function $l_{diffcnn}$ depend on V, the set of all variables in the network, f_{nn} that is defined by the network graph G, and D, a batch of size B of training data.

4.4 Training stop criteria

It was shown in [6] that the fully convolutional architecture with the global average pooling layer has little tendency to overfit, even on small datasets. Many very small datasets are also not big enough for a validation set. This also applies to the architectures generated and datasets used in this thesis. Therefore, we run gradient descent for a fixed number of epochs n_{epochs}, generating a new model

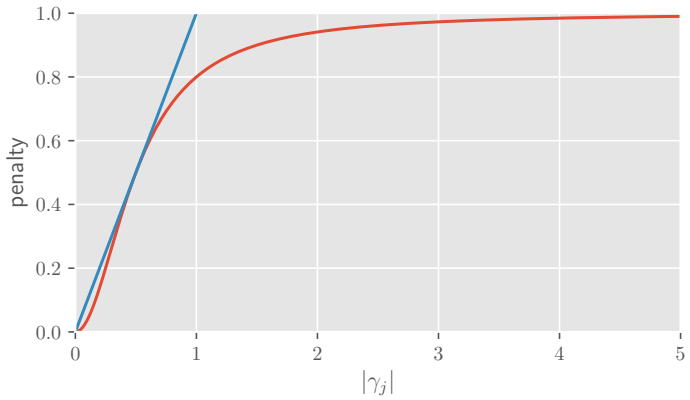


Figure 6: The Weigend penalty function for a single channel scale γ_j for $\kappa = 0.25$ compared to a linear penalty function

$m \in M$ after every epoch. We choose the model $m_{best} \in M$ with minimum loss on the training set T

$$m_{best} = \underset{m \in M}{\operatorname{argmin}} \left[l_{diffcnn}(m, T) \right] \tag{22}$$

The model m_{best} can then be evaluated on the test set. An epoch is defined as one training pass through the entire training set T. A model is defined by $m = (V, G)$ where V is the set of all variables, including weights, biases, and scales while G is the graph that defines the network's architecture and can also be expressed as the composite function f_{nn}. Additionally, in order to save computational resources, we can stop the training procedure if the minimum loss obtained until the current step does not improve for $n_{stagnant}$ steps.

Having defined the stopping criteria, we can now outline the entire algorithm for the DiffCNN in algorithm 1. The mutation procedure will be defined in the following subsection and for simplicity, we use stochastic gradient descent (SGD) with mini batches in the algorithm description. In practice, we employ Adam [29] as later described in section 5. Nevertheless, any other optimizer can be used in this context as well.

4.5 Mutation phases

Gradient descent can control γ_j to activate or deactivate existing channels in any layer. In order to unlimitedly grow or prune channels and change the depth of the network we run mutations on the neural network, directed by gradient descent. Instead of growing or pruning the current architecture after every gradient descent step, we train the weights using our regularized loss function from Equation 21

Algorithm 1: The complete algorithm for mutating architectures

Data: n_{epochs} the number of epochs to train for
n_{batch} the size of the batches
$s_{training}$ the number of training steps between mutations
$l_{diffcnn}$ The loss function as defined in Equation 21
T the training set

Result: A CNN defined by the set of variables V and the architecture graph G

V, G ← initial architecture;
L ← empty list ; `// List that holds history of generated architectures`
step counter ← 0;
foreach $e \in [\ 1\ ..\ n_{epochs}\ ]$ **do**
  $T \leftarrow shuffle(T)$;
  $loss \leftarrow 0$;
  $T_{batched} \leftarrow$ batches of size n_{batch} of T;
  foreach $D \in T_{batched}$ **do**
    $loss \leftarrow loss + \frac{l_{diffcnn}(V,G,D)}{|T_{batched}|}$;
    `// Gradient descent`
    $V \leftarrow sgd(V)$;
    `// Mutate every` $s_{training}$ `steps`
    if *step counter* mod $s_{training} = 0$ **then**
      $V, G \leftarrow mutate(V, G)$;
    end
    step counter ← step counter + 1;
  end
  $L \leftarrow L \cup \{(loss, V, G)\}$;
end
return $\text{argmin}_{l \in L}\, l_{loss}$;

Procedure sgd(V)

Input : variables V
Output: updated variables V'

$V' \leftarrow \{\}$;
foreach $v \in V$ **do**
  $g \leftarrow \frac{\partial}{\partial v} l_{diffcnn}(V, G, D)$;
  $v' \leftarrow v + \lambda g$; `// Or a more sophisticated optimizer than SGD`
  $V' \leftarrow V' \cup \{v'\}$;
return V';

for a fixed count of steps $s_{training}$. This method has several advantages compared to mutating every step. Firstly, γ_j for each output channel j can arrive at an equilibrium of usefulness vs penalty contribution after $s_{training}$ steps have passed.

Secondly, it allows us to initialize the scales γ_j to a common default of $\gamma_j = 1$ instead of having to initialize them at the deletion threshold t_{del}. This enables much better training as later discussed in subsection 4.7. Thirdly, common deep learning frameworks such as TensorFlow [30] need to define a fixed computational graph in advance. Rebuilding the graph for mutating the CNN is time expensive and should not be done every step.

For growing or pruning the number of channels in a convolutional layer i we test whether $|\gamma_j| < t_{del}$ for all channels j. Let $n_{inactive}$ be the number of channels that satisfy this condition. Also, as previously introduced, a threshold $t_{inactive} = 1$ is defined. If $n_{inactive} > t_{inactive}$ we prune $n_{inactive} - t_{inactive}$ of those channels. Likewise, if $n_{inactive} < t_{inactive}$ we add new channels to the layer. While adding a single channel may be sufficient, we add $O_{add} >> 1$ for two reasons. Firstly, because mutations are not performed every step, adding only a single channel would lengthen training considerably. Secondly, every time we mutate, another layer can also be created. If the number of channels does not grow quickly enough, more layers are favored over shallower networks that are easier to train.

So far we described only how to modify the number of output channels in a convolutional layer. We now extend the concept to expanding and reducing depth. In subsection 4.1 Figure 4 we defined the initial architecture. Whenever we add a new layer we need to achieve three things: Firstly, the new layer should enable to learn more complex patterns, therefore learning a representation based on a previous representation. Secondly, the existing learned weights should be reused instead of undoing the learning progress. Thirdly, the new layer should only be used if it reduces the cross entropy enough to make up for the gain in penalty. Thus, the optimization objective for gradient descent must allow to choose between keeping the old architecture or using the newly created layer. We satisfy these requirements by introducing a new skip connection every time we add a new layer, as seen in Figure 7. A skip connection feeds the outputs of a convolutional layer directly to the global pooling layer. In effect, before global pooling, we concatenate the output channels of all convolutions to a matrix $M \in \mathbb{R}^{B \times L \times \hat{O}}$ where $\hat{O}$ is the number of total output channels in the network. After global pooling, we yield $M_{pooled} \in \mathbb{R}^{B \times \hat{O}}$ which can be fed to the softmax layer. In consequence, the softmax layer can linearly combine all mean reduced channels from all convolutions.

Having defined the structure the architecture can take the form of, we need to define the rules that govern when to add and delete a layer. In order to yield a framework that has a consistent methodology we aim to define those rules similar to adding or removing channels. Deleting a layer is simple to derive. If all channels in a layer are inactive, it means that the layer does not serve any purpose in the network and can, therefore, be deleted. Because all the following convolutional layers depend on the deleted layer's outputs, we can delete those as well. For

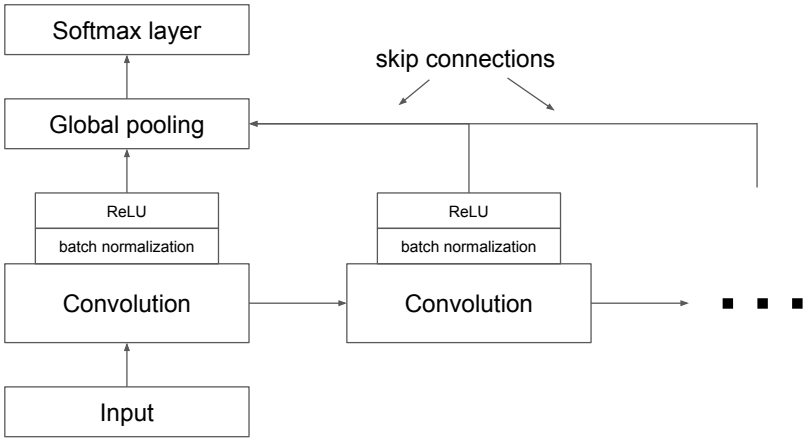


Figure 7: How the architecture can grow and shrink in depth using skip connections

instance, if we have n convolutional layers in the network and in layer $n-1$ all output channels are below the deletion threshold, then we can delete both layer $n-1$ and layer n. A new layer is added whenever the current top-most convolutional layer $d-1$ has only active output channels, along with the previously defined extension of output channels. Those rules are built under the assumption that adding a single new channel or layer is sufficient to achieve any gain in cross entropy if adding an arbitrary count of new channels and layers can achieve a gain. While this may not be strictly true for all problems, we show experimentally that this is a reasonable assumption.

It is left to discuss the changes to network variables that are required whenever a new channel or layer is removed or added. A convolutional layer with batch normalization has three variables: The kernel w, scales γ and the bias β. Additionally, we keep track of the running averages of the mean μ_{EMA} and variance σ_{EMA} per channel for evaluation with batch normalization. Whenever we add C_{added} output channels to layer i we expand the output channel axis of w by C_{added} and randomly initialize the new values with the weight initializer that was also used for the initial architecture. Furthermore, the vectors γ and β need to be expanded by C_{added} dimensions as well, with γ being initialized to one and β being initialized to zero. The running averages μ_{EMA} and σ_{EMA} are enlarged by C_{added} and initialized to zero. As the number of output channels of layer i now has changed we need to change the input channels of all dependent operations. In the case of our architecture that is the convolutional layer $i+1$, if it exists, and the softmax layer d that linearly combines all channels. For that, the weights of the softmax layer are expanded in its first axis (number of input neurons) by C_{added} and randomly initialized. The input axis possibly has to be expanded at a specific position instead of the end, depending on the concatenation of the output channels that make up the operation's input. The weights $w^{(i+1)}_{convolution}$ are expanded in its input channel

axis and randomly initialized. Likewise, whenever output channels are removed the variables need to be reduced in size using a mask accordingly. New layers and removed layers affect the channel input axis, channel output axis and the softmax weights in a similar manner.

Procedure mutate(V, G)

Input : variables V and network graph G
Output: updated variables V and network graph G

`// Mutate for every layer` i `in the network of depth` d `(excluding input and softmax layer)`

foreach $i \in [\ 2\ ..\ (d-1)\ ]$ **do**
  `// Obtain all channels that are below the deletion threshold`
  $inactive^{(i)} \leftarrow channels(layer_i)$ where $|\gamma^{(i)}| < t_{del}$;
  `// Prune or grow a channel`
  if $|inactive^{(i)}| > t_{inactive}$ **then**
    prune $|inactive^{(i)}| - t_{inactive}$ channels from layer i;
    update weights, biases, and scales of this layer i and weights of all consuming layers ; `// consuming layers are the softmax layer and layer` $i+1$ `if it exists`
  if $|inactive^{(i)}| < t_{inactive}$ **then**
    add O_{add} channels to layer i;
    update weights, biases, and scales of this layer i and weights of all consuming layers ; `// consuming layers are the softmax layer and layer` $i+1$ `if it exists`
  `// Delete layer if no channels left`
  if $|channels(layer_i)| = t_{inactive}$ **then**
    delete this layer i and all following layers $[\ i+1\ ..\ d-1\ ]$ except the softmax layer;
    update weights of softmax layer;

`// Add a new layer if all channels in last convolutional layer are used`

if $|inactive^{(d-1)}| = 0$ **then**
  add new layer after layer $d-1$;
  add skip-connection from layer $d-1$ to softmax layer d;
  update weights of softmax layer;

return updated V and G

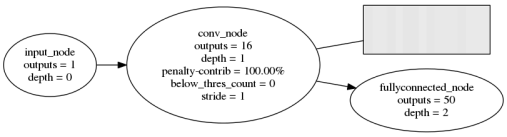
4.6 Illustration of the DiffCNN

The entire process is visualized in Figure 8. The CNN is rendered in a simplified version presented as a graph. We omit the batch normalization, activation functions, concatenation operations and global average pooling. The previously printed Figure 7 shows where those concepts are used. Every convolutional layer has a rectangular rendering with vertical bars attached. Those bars visualize the penalty induced by every output channel from left to right as defined in Equation 18. A black vertical bar represents a fully switched off channel, while gray to white bars describe channels that are in use. Figure 8a shows the initial architecture after very few gradient descent iterations. Therefore, the scales γ are still very close to their initialization of one. After $s_{training}$ gradient descent steps in Figure 8b, before mutating the architecture, all of the channels are clearly in use, some having bigger scales than others. In Figure 8c we mutated the architecture. The first convolution used all its channels, thus we raised the number of channels by C_{add} and added a new convolutional layer that used the first convolutional layer's output as input. Additionally, a skip connection was added to continue training smoothly. At some point, as shown in Figure 8d, the last convolution's channels are not further minimizing the cross-entropy, therefore some of them become deactivated and are visualized in black. During the next mutation in Figure 8e those channels are pruned and no additional layer is added.

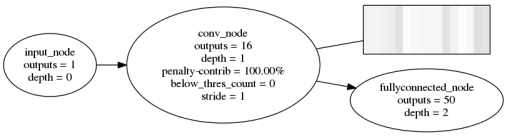
4.7 Switched-on initialization

Initializing the scales to the deletion threshold t_{del} as originally done with the L1-norm of outgoing weights in [9] leads to a bad initial configuration for gradient descent. The small scales impede the optimization objective so that output channels tend to get deactivated or the magnitudes of γ_j are generally very small. This phenomenon will be closer inspected in section 5. Furthermore, a lot of research has been put in good weight initialization, with respect to probability distribution, magnitude, and variance. One, as of to date commonly used initialization, is the Xavier initialization [31]. It adapts its distribution based on the number of incoming connections and outgoing connections. We employ the Xavier uniform initialization.

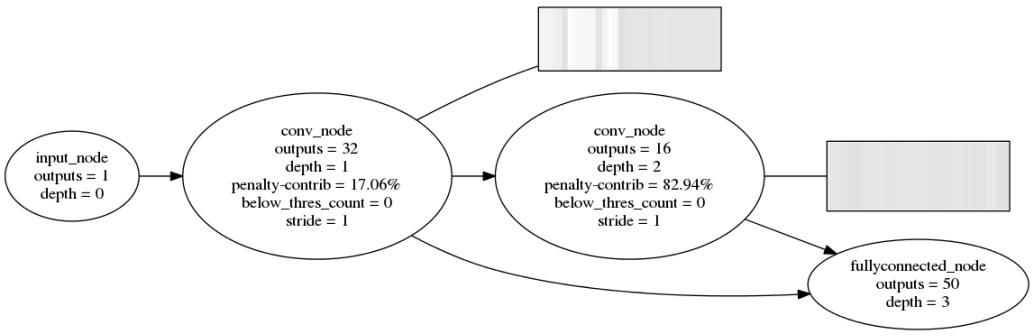
In order to take advantage of that research it is favorable to initialize the scales to one instead of t_{del}, as usually done with batch normalization. As a result, we run several gradient descent steps $s_{training}$ before pruning and growing, as previously described in subsection 4.5. During those steps, the scales can adapt to a stable configuration.



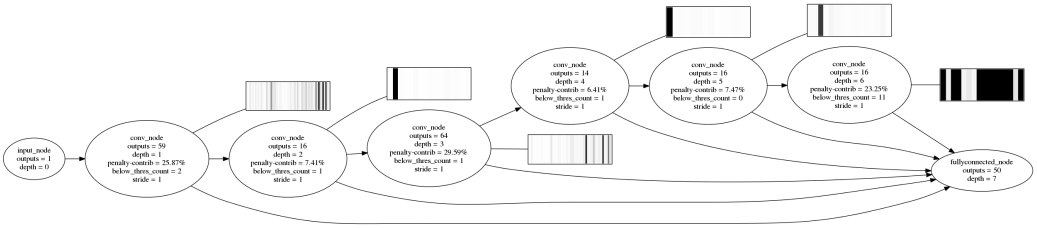
(a) The intitial architecture after only a few gradient descent iterations



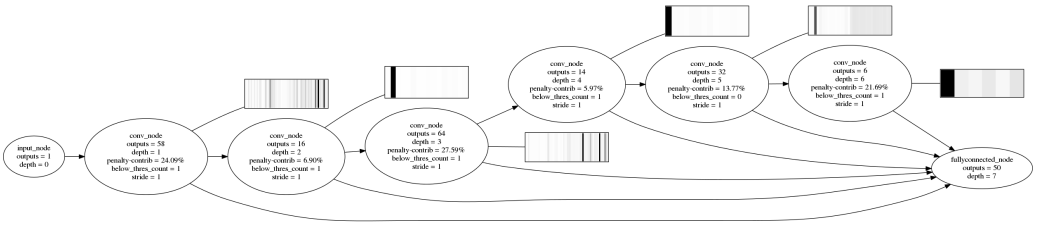
(b) All of the channels are in use, no mutation has been performed yet



(c) The mutation added new channels to the first convolution, added a new layer and a new skip connection



(d) Most of the channels in the last convolution are deactivated



(e) The next mutation pruned the deactivated channels in the last convolution and did not add a new layer

Figure 8: A visualization of the changing architecture

4.8 Two-phased learning

Experimental evaluation showed that it takes many gradient descent iterations to update the scales γ to a stable configuration. This process takes much longer

compared to updating the weights w according to the optimization objective. One solution to this problem is to raise the learning rate. While this updates the scales quickly, it impedes proper updates of w leading to a non-converging cross entropy. Another option is to raise the penalty factor α, therefore weighing the penalty stronger and increasing the gradient of the scales γ. But this imbalances the relative importance of cross entropy and reduction of complexity through the penalty.

We introduce a new concept to solve this issue. If we update the scales γ independently from the weights w with different learning rates we achieve both a convergence in γ and a convergence in w. Separating the updates of both variable sets would lengthen training time as progress is repeatedly undone. Instead, the first phases trains γ and w jointly, while the second phase only updates γ. Now the question arises when each phase needs to be run. Updating γ requires that w is in effective ranges already, thus we can not run the second phase right after a mutation. Running the second phase before the next mutation and after the first phase has completed may not arrive at a local minimum of the loss function due to inaccessible weights w. Therefore we employ the following strategy: First gradient descent updates γ and w jointly, then only γ, and finally another phase of jointly training.

The entire cycle is shown in Figure 9. We specify the length of the γ updating phase as a fraction r_{scales} of the total training steps $s_{training}$ up until the next mutation. Additionally, the learning rate for the γ phase is given by λ_{scales}. In procedure sgd-with-training-phases we highlight how the training algorithm needs to be adapted. In section 5 we evaluate how the two-phased learning improves training.

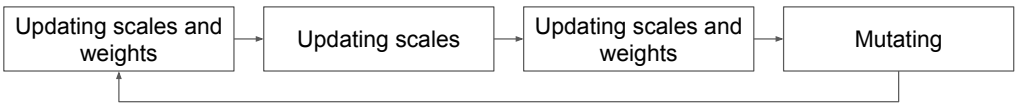


Figure 9: The several different phases in training

4.9 Implementation

Commonly deep learning frameworks such as TensorFlow [30] are not designed to change their graph structure between training steps. Therefore, the computational graph is rebuilt after each mutation from an architecture specification that is held separately. Figure 10 shows how this specification is modeled as a directed acyclic graph. Every *Node* produces an output tensor that can then be consumed by all children. *VariableNode*s represent convolutional or fully connected layers and can specify an additional penalty that is imposed due to their scales γ. The sub

Procedure sgd-with-training-phases(V, step counter)

Input : variables V, step counter
Output: updated variables V'

$V' \leftarrow \{\}$;
`// Select a subset of variables and learning rate depending on the current training phase`
if *is-joint-training-phase (step counter)* **then**
  $S \leftarrow V$;
  $\lambda' \leftarrow \lambda$;
else
  $S \leftarrow V_\gamma$ where $V_\gamma \subset V$; `// only use scales` γ `of all variables`
  $\lambda' \leftarrow \lambda_{scales}$;
foreach $v \in S$ **do**
  $g \leftarrow \frac{\partial}{\partial v} l_{diffcnn}(V, G, D)$;
  $v' \leftarrow v + \lambda' g$; `// Or a more sophisticated optimizer than SGD`
  $V' \leftarrow V' \cup \{v'\}$;
return $V' \cup (V - S)$;

classes *ConvolutionalNode* and *FullyConnectedNode* have attributes like kernel size and the current number of channels or neurons. Each node type is responsible for generating its respective part of the computational graph and maintaining integrity after network mutation. This includes adding concatenation operations to merge inputs or adapting variables to deal with a changed size of inputs or outputs. The entire structure integrates with the TensorFlow variable saving system to restore from disk if training needs to be continued or evaluation is invoked. Additionally, visualizations of the architecture such as in Figure 8 can be generated at any point in time and inspected in TensorBoard with many other statistics in real-time. A command line interface with a wide range of parameters enables exploration of many different ideas that can be used to govern the DiffCNN. The application has the capability to run the entire training and evaluation on the GPU. Along the DiffCNN, it provides a reimplementation of [6] using the directed acyclic graph as a specification that can not be mutated. To enable reproducibility the entire source code is available at `https://github.com/timediv/diffcnn`.

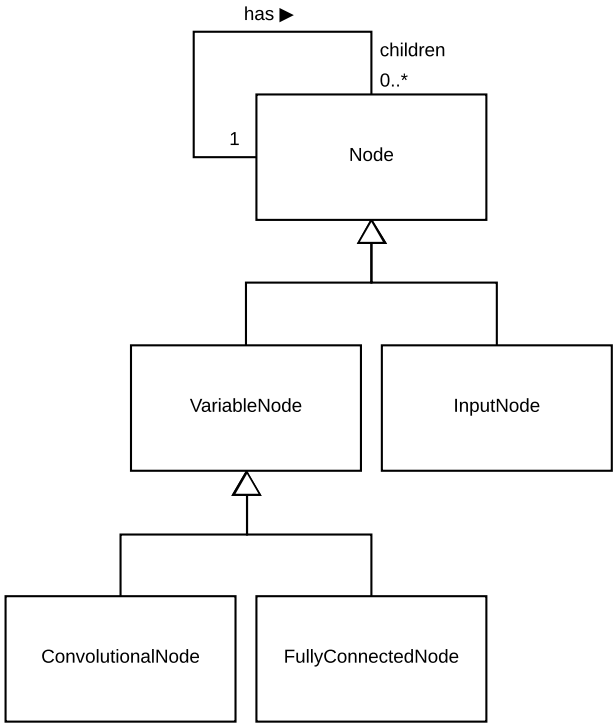


Figure 10: The neural network architecture is specified by a simple directed acyclic graph modeled as above

5 Experiments on UCR datasets & Discussion

In this section, we will evaluate the DiffCNN on the UCR datasets [10] and analyze the advantages and limitations of the approach. The UCR archive is one of the most prevalent repositories for TSC featuring 85 univariate time series datasets. It allows a detailed comparison to existing deep learning, feature extraction, distance based and ensemble based approaches. In the following, we begin with defining the experiment settings, including training, all hyperparameters and preprocessing. Then, we present and discuss the results on the UCR datasets. We proceed with analyzing how the network complexity is controlled by gradient descent and whether the optimal architecture has been chosen. We show that both the switched-on initialization and the two phased learning are necessary to achieve good convergence and suitable architectures. We investigate good choices of the penalty factor α and how training with mutations generally behaves, particularly in contrast to the FCN's training. Finally, complexity and scalability of CNNs and the DiffCNN are mentioned and the limitations of the approach are determined.

5.1 Experiment settings

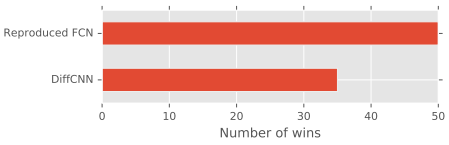
For the joint training phase as described in subsection 4.8, we employ the Adam optimizer with its default values from [29]. Weights are initialized Xavier uniformly as introduced by [31]. An overview over all required hyperparameters for the DiffCNN is given in the Appendix A in Table 2, including a description, their default values and how to set them if dataset dependent treatment is required. On the wide range of UCR datasets, no special parameter tuning is necessary and therefore all datasets are used with the same values. Conclusively, the default parameters are sufficient to handle a wide variety of datasets.

The UCR datasets are already split into training and test sets and, therefore, the results can easily be compared to other algorithms. A minimal amount of preprocessing is required to apply the DiffCNN to all 85 datasets. We only z-normalize the input using the training set mean and variance prior to feeding it to the CNN.

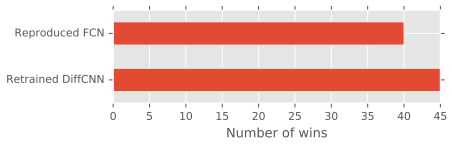
5.2 Performance evaluation

We report the results on all 85 UCR datasets. Compliant with the competitors we adopt the testing error e, given by

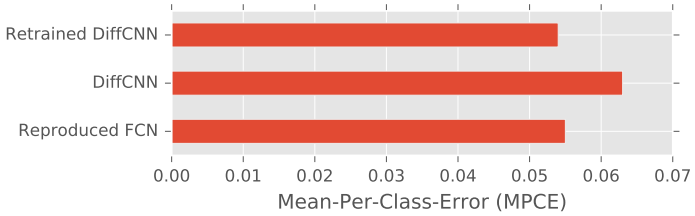
$$e = 1 - p_{accuracy} \tag{23}$$



(a) Reproduced FCN vs DiffCNN



(b) Reproduced FCN vs Retrained DiffCNN



(c) MPCE on the DiffCNN, the retrained variant and the reproduced FCN

Figure 11: Comparison with the reproduced FCN on all 85 UCR datasets

The accuracy is calculated on the entire test set for the architecture and weights with minimal training loss as described in subsection 4.4.

Figure 11 compares the DiffCNN with a reproduced version of the FCN. Both the source code provided[2] by the FCN's authors [6], as well as our own implementation, yields slightly worse results than reported in their publication. Due to the focus on mutating architectures by differentiation, we will compare the fixed architecture of the reproduced FCN with the dynamic architecture of the DiffCNN. Later, we add a comparison with the published results of the FCN and other related TSC algorithms. We will use our own implementation of the FCN for a fair comparison with the DiffCNN.

We present two types of measures in Figure 11. The first measure is the mean per class error (MPCE). It was originally defined [6] for a better comparison of competitors on the UCR archive. It is given by

$$PCE_k = \frac{e_k}{c_k} \qquad MPCE = \frac{1}{K}\sum_k PCE_k \tag{24}$$

where k refers to each dataset and K is the number of all datasets. The error on the dataset k is denoted as e_k with c_k being the number of classes in that particular dataset. The MPCE is evaluated for each classifier over all datasets. The second measure is the number of wins, the number of times a classifier outperforms all

[2] `https://github.com/cauchyturing/UCR_Time_Series_Classification_Deep_Learning_Baseline`

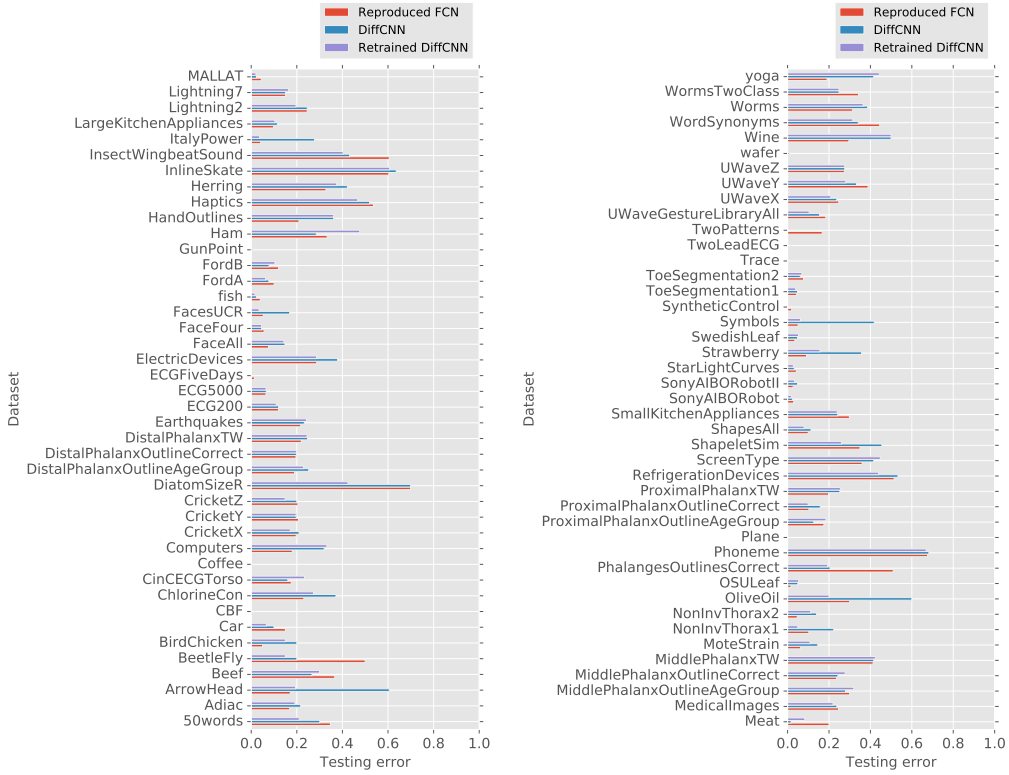
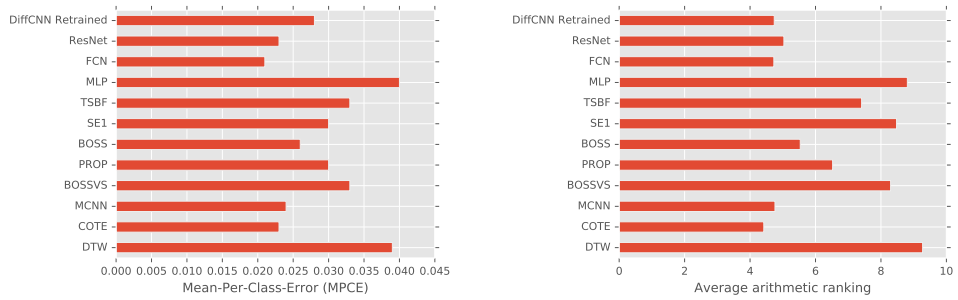


Figure 12: Testing error of the DiffCNN, compared to the FCN, the state of the art in TSC, while employing a fixed CNN architecture

other classifiers in the comparison. The figures 11a and 11c show that even though the DiffCNN automatically learns its architecture it performs only about 1% worse in the MPCE measure while having slightly fewer wins over all datasets. When retraining the DiffCNN using its final architecture by reinitialization with random weights, biases, and scales we outperform the FCN in the number of wins. Figure 12 shows the testing error on all 85 datasets for the DiffCNN, the retrained version and the reproduced FCN. The exact values can be found in the Appendix A in Table 3.

Finally, we compare competitors from section 2 with the retrained DiffCNN. We use a selection of 44 UCR datasets that was originally used in other publications in order to achieve comparability. We did not reproduce their approaches but use the publication values from [6]. Figure 13a shows the MPCE for all classifiers. The DiffCNN is competitive both in terms of its MPCE as well as its mean ranking over all datasets in Figure 13b. The Appendix A contains the testing errors of all 44 datasets in Table 4.



(a) MPCE for each classifier on all 44 datasets

(b) Mean ranking on all 44 datasets

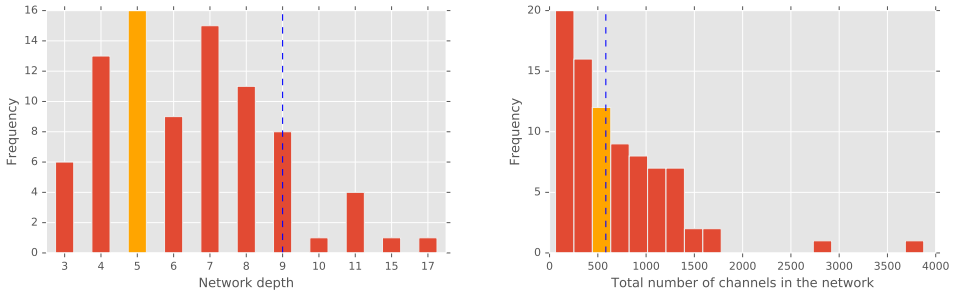
Figure 13: Comparison on 44 UCR datasets with published results from related work

5.3 Adapting network complexity

The DiffCNN is designed to change the network complexity in terms of network depth and the number of channels per convolutional layer automatically. Naturally, different datasets vary in their inherent complexity. Some datasets are easily classified because of simple separability of classes. Others contain complex patterns that require the network to have several intermediate representations and therefore a deeper architecture.

We investigate whether the introduced algorithm does adapt to the complexity of the respective dataset. If so, the final architecture with the lowest training loss would have different depths and number of total output channels. We recorded the total number of channels and the depth for all UCR datasets when the training loss was the lowest. Figure 14 shows that both depth and the total number of channels vary considerably over the datasets. Very simple datasets like the *WormsTwoClass* dataset only required a depth of 3 to achieve a 9.4% better accuracy than the deeper handcrafted FCN architecture with depth 5. A depth of 3 embodies an input layer, a single convolution, and the softmax layer. The final architecture for this dataset can be seen in Figure 15. In contrast to that, the *50words* dataset profits from a much deeper architecture of depth 9, also yielding a better accuracy compared to the FCN baseline. Figure 16 shows this rather big architecture.

When comparing the complexity of the DiffCNN architectures and the handcrafted FCN architecture, it is interesting to observe in Figure 14b that the median number of channels of the generated architectures is quite similar to the fixed number of 512 channels in the case of the FCN. This indicates that the handcrafted number of channels is a good compromise for the UCR datasets. On the other hand, as seen in Figure 14a, the DiffCNN has the tendency to produce deeper architectures



(a) Histogram of depth over all UCR datasets

(b) Histogram of total number of channels over all UCR datasets

Figure 14: Final architectures with lowest training loss have different complexities depending on the dataset — FCN handcrafted architecture highlighted in orange, blue dashed line visualizes the DiffCNN's median

than the FCN's depth of 5. A possible explanation for this observation is that layers are added iteratively and therefore can not be optimized jointly right from the beginning of training. This may lead to non-optimal convolution kernels that could, in theory, be condensed to fewer layers.

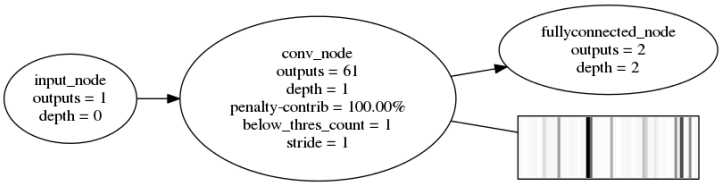


Figure 15: A very simple architecture for the *WormsTwoClass* dataset is sufficient to outperform the FCN baseline

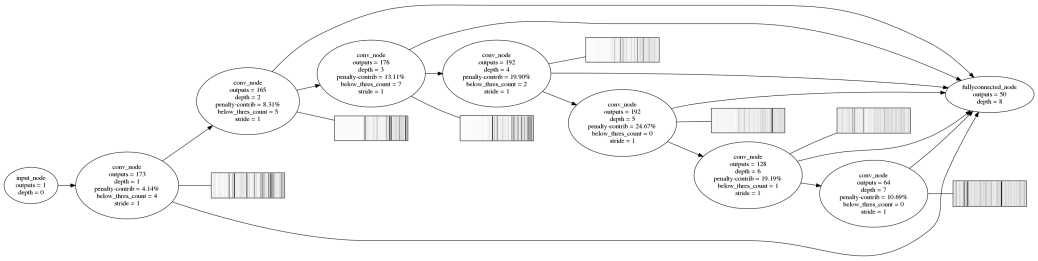


Figure 16: The *50words* dataset profits from a much deeper architecture in order to classify the 50 target classes with high accuracy

While we already showed that the depth and number of channels vary between datasets and can lead to better accuracies compared to a fixed architecture, we need to investigate whether the final architecture is the optimal choice. For that, we train several different architectures from scratch with varying depths and compare

it with the DiffCNNs final architecture. For this experiment, we use the *Car* dataset because it produces a medium network depth compared to other datasets. We can thus analyze how shallower and deeper variants perform. When running the DiffCNN the final architecture has a depth of 7 and the mean of the number of channels is 150. Therefore, we ran several CNNs (DiffCNN without mutation) on the *Car* dataset with a fixed number of 150 channels per layer and a depth varying from 3 to 9. Figure 17 shows that a depth of 6 or 7 yields the best possible test accuracy on the dataset. The automatically picked depth of 7 by the DiffCNN, therefore, is a good choice for the dataset. This shows that the mutation phases produce networks with reasonable complexity.

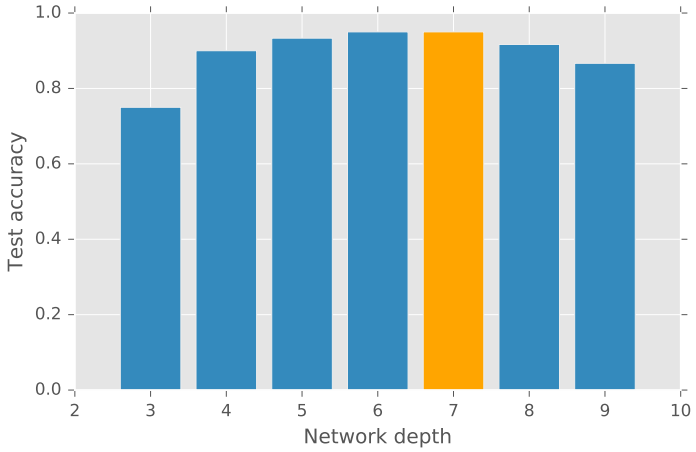
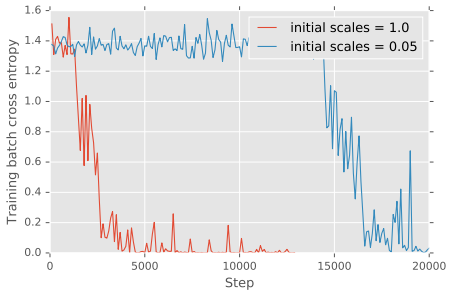


Figure 17: We varied the number of layers with a fixed channel count of 150, the DiffCNN's depth is highlighted in orange

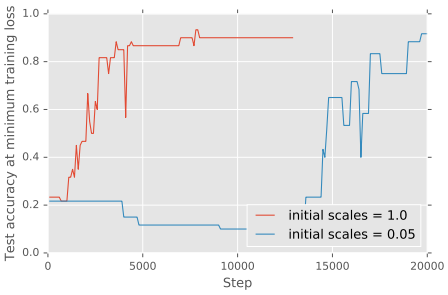
5.4 Necessity of switched-on initialization

In subsection 4.7 we claimed that optimization is impeded when the scales γ are initialized to the deletion threshold t_{del}, as originally done with the weights in [9]. In this section, we experimentally show that this is the case. We trained two variants on the *Car* dataset, one with γ initialized to $\gamma_{init} = 1$, the other initialized to $\gamma_{init} = t_{del} = 0.05$. In Figure 18 we observe that an initialization at the deletion threshold leads to a long period of being stuck in a local minimum. Only after $15,000$ steps of training, this minimum is left behind and the loss function is properly optimized. The training that has been initialized with $\gamma_{init} = 1$ has already converged at that point and training has terminated.

The same problem applies to the optimization of the architecture. In Figure 19 we can observe that the architecture has a too low complexity for long periods of

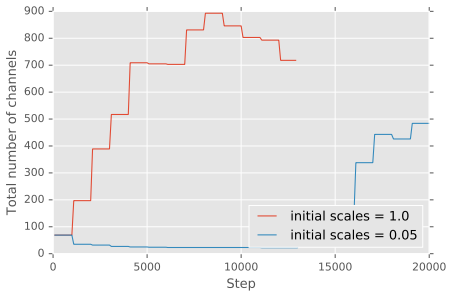


(a) Optimization of the cross entropy

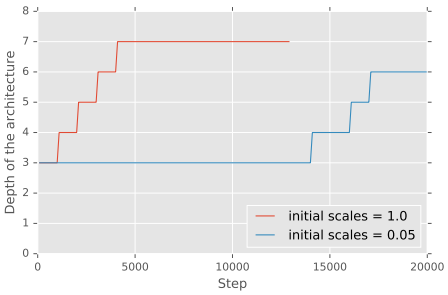


(b) Test accuracy at minimum training loss up until current step

Figure 18: Setting the initial scales to the deletion threshold t_{del} leads to a long period of being stuck in a local minimum



(a) The total number of channels and neurons in the network



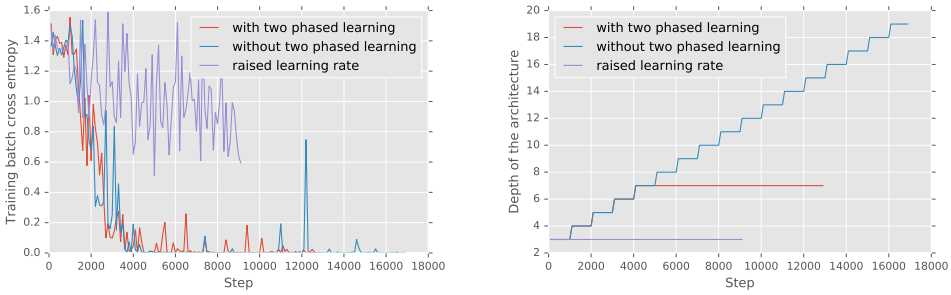
(b) The depth of the network

Figure 19: Setting the initial scales to the deletion threshold t_{del} is reflected in an architecture that is too simple for a long period of time

time. Right in the beginning the first mutations reduce the channels to a very small number before the network recovers and starts adding channels and layers again.

5.5 Necessity of two-phased learning

In subsection 4.8 we proposed to use two kinds of learning phases. The first phase trains weights, biases, and scales jointly, while the second phase only changes the scales. The second phase is embedded within the first phase with a ratio r_{scales} of all training steps $s_{training}$. We reasoned that we require this separation in order for the switches to converge fast enough. In this section, we will experimentally show the necessity of this approach.



(a) Cross entropy optimization

(b) Growth of the network depth

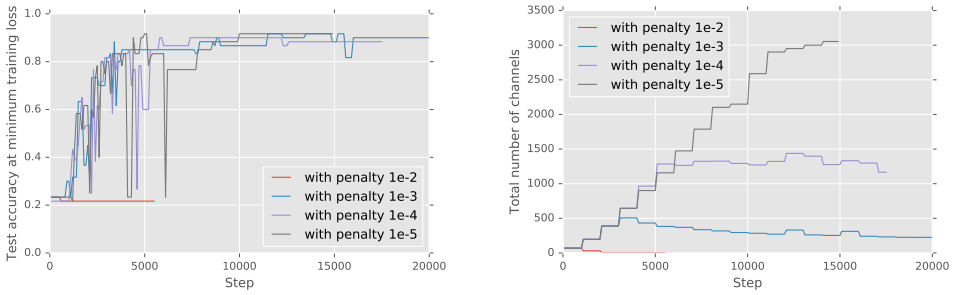
Figure 20: Only the two phased learning can achieve both a good cross entropy convergence, as well as a stable architecture

We analyze the effects of always optimizing all variables jointly. In Figure 20b we observe that keeping the same learning rate of $\lambda = 10^{-3}$ for the entire training results in an architecture where the scales never fall below the deletion threshold and the networks just keeps growing indefinitely. Raising the learning rate to $\lambda = 0.05$ for the entire training inhibits a convergence of the cross entropy and also keeps the architecture from growing. Figure 20a shows that in the form of an oscillating cross entropy that is barely reduced over time. With the introduction of a joint learning phase with a learning rate of $\lambda = 10^{-3}$ and a separate scales learning phase with a learning rate of $\lambda_{scales} = 0.05$ we achieve both a cross entropy convergence, as well as a stable architecture.

5.6 Experimental choice of penalty factor

The DiffCNN uses a previously in Equation 21 defined loss function. This function has a parameter α that weighs the penalty against the cross entropy. Therefore, the network complexity assumably depends on this penalty factor α. To visualize the effects, we varied the several orders of magnitudes of penalty factors on the *Car* dataset. On all datasets, the cross entropy after 100 iterations is in the order of 0.1 to 2.0.

Figure 21b shows that indeed the complexity of the final architecture is affected by this factor. It is now in question whether this complexity has a major impact on the final test accuracy. From Figure 21a and Figure 21b it is obvious that a high penalty factor of $\alpha = 10^{-2}$ impedes both the growth of the architecture, as well as the final test accuracy at the lowest training loss. Whereas even though the complexity of the architectures for $\alpha \in [10^{-3}, 10^{-5}]$ differ significantly, the test accuracy barely differs. We can conclude that even with more architecture complexity there is little tendency of overfitting. By evaluation on all UCR dataset



(a) A high penalty factor results in low test accuracy while smaller values lead generally to good test performance

(b) The penalty factor considerably affects the total number of channels over time

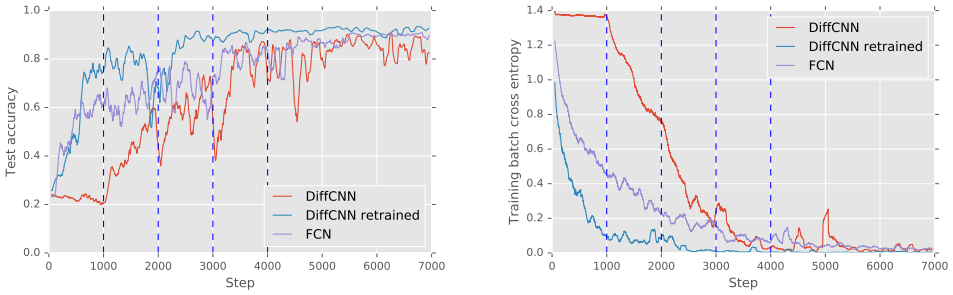
Figure 21: Varying the penalty exposes how test performance and network complexity are affected

we settled for a common penalty factor of $\alpha = 10^{-4}$ that generally works well on a wide range of datasets.

5.7 Training behavior and duration

In this section, we perform experiments on the *Car* dataset to expose how training behaves for the DiffCNN. We analyze distinct features and compare it with the FCN's training. Furthermore, we inspect how training differs when the DiffCNN's final architecture is taken and variables are reinitialized and retrained.

We logged test accuracy and training batch cross entropy after every step of training. Moreover, test accuracy at the minimum training loss over all previous steps was tracked. For better visual presentation a mean filter of filter size 100 is applied to all data in Figure 22. Figure 22a shows how the test accuracy changes during training. The mutations that added a new layer to the DiffCNN network every $s_{training} = 1000$ steps are marked with a dashed line in the plot. Until the step $s = 1000$, we notice that the test accuracy does not improve. Only with the first mutation, the required new layer is added and the total depth of the network is raised. Shortly thereafter the test accuracy improves significantly. With every mutation that adds a new layer, the test error can improve until $s = 5000$. In the followed training the test accuracy stagnates and no more layers are added. Several distinct features can be observed in the plot. Firstly, after every mutation, we can observe a general improvement in test accuracy until convergence. Secondly, after a mutation, the test accuracy often drops shortly before it quickly rises again. This can be explained by the scales γ that we initialize to one, shortly disrupting



(a) The test accuracy after every training step

(b) The cross entropy on the current training batch

Figure 22: The training behavior of the DiffCNN, retrained version and the FCN on the *Car* dataset

the network's predictions on the test set. In Figure 22b it can be observed how the cross entropy behaves for each training batch. In this plot, after a mutation, no sudden rises of the cross entropy can be observed. We can conclude that while the general test performance is inhibited by the mutation, the network can quickly re-optimize the current training batch. Additionally, from Figure 22b it is obvious that new layers are added up until the cross entropy is very close to zero.

In Figure 22 we also included the FCN's training and DiffCNN's retraining behavior in all plots. When retraining the DiffCNN with its final depth of 7 we no longer need to mutate the architecture. Therefore, the test accuracy quickly rises in just the first $s = 1000$ steps. Nevertheless, it takes several thousand steps more to converge in terms of test accuracy. The same behavior can be observed in terms of change in cross entropy. In contrast to that, the FCN architecture with its smaller depth of 5 takes longer to reach a low cross entropy and high test accuracy. In Figure 22b it can be observed that it actually takes about just as many steps for the FCN to converge compared to the DiffCNN that learns its architecture during the process. If we decide to additionally retrain the DiffCNN's final architecture we end up with about 1.5 times more steps required. Apart from this, there is additional time lost by rebuilding the graph during mutation and the generally deeper architectures that are produced. We measured[3] the training time of both the DiffCNN and the FCN to determine how much more time is needed for each step. In the mean, each step of the DiffCNN takes 3.3 times longer while the median is at 3.0.

Figure 23 shows the test accuracy at minimum training loss for all aforementioned three training runs. The general trend is similar to the current test loss from Figure 22a. The same applies to the final test accuracy at the end of training.

[3] on a GeForce GTX Titan X graphics card

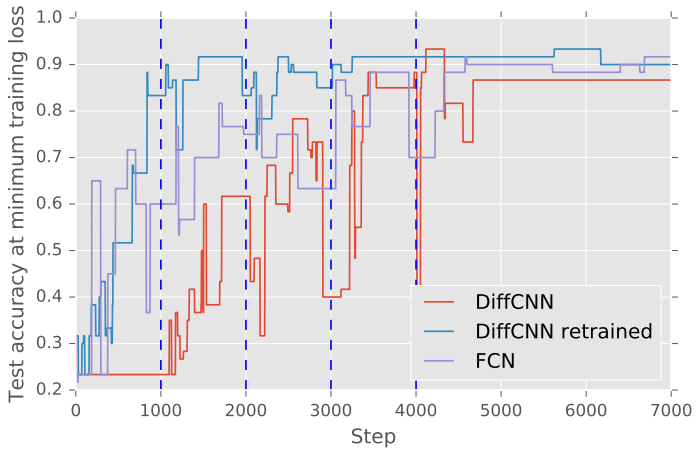


Figure 23: The test accuracy at the minimum training loss after every step for the DiffCNN, retrained version and the FCN

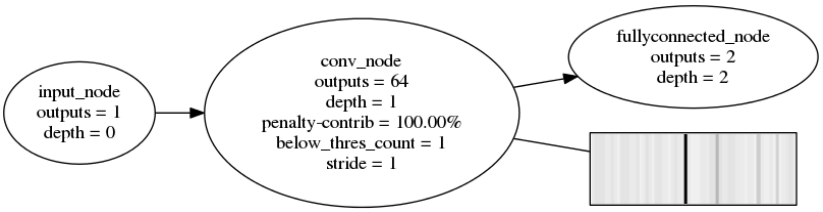
5.8 Complexity and scalability

Generally, the same complexity and scalability apply to the DiffCNN as for any other CNN. A complexity analysis is performed in [32]. We now shortly inspect how the number of samples, classes, and length of the time series impacts training. The training time until loss convergence does not depend on the number of samples. Adding more samples would not slow down training but improve generalization performance [33]. Furthermore, the UCR archive includes datasets containing two to 60 different classes. Neither the lower nor the upper end poses a challenge to CNNs or the DiffCNN, making the approach fairly scalable in terms of the number of classes. The time series length was limited to 2709 in the case of the UCR archive. The length of the time series affects runtime quadratically [32] but does not harm the DiffCNN's classification performance as long as the kernel size is adapted properly to the size of patterns in the series.

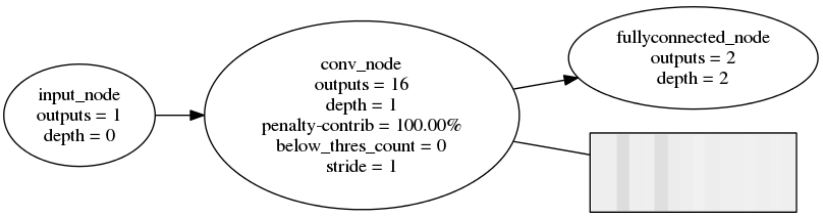
5.9 Limitations of the DiffCNN

While the DiffCNN has competitive test set performance it also fails on some datasets. In this section, we investigate general limitations and specific failure cases of the approach. One major problem that has not been addressed with the DiffCNN approach is the choice of the kernel. While this parameter is much easier to set than the network depth or number of channels, it still needs to be picked manually according to the user's domain knowledge. Furthermore, the penalty factor α is not strongly dataset dependent but care must be taken to not choose

too big values that restrict architecture growth. The overall depth of the average network was shown to be slightly raised compared to the well performing fixed architecture of the FCN due to incremental architecture expansion. While the increased depth does not harm final test accuracy it does slightly raise hardware requirements during training and testing. Additionally, no striding was used in the DiffCNN architectures. Striding may reduce hardware requirements by shrinking the spatial extent of the feature maps and raising the receptive field of the following neurons without growing the kernel size [24].



(a) Not all channels are used and the depth will therefore not be increased during the next mutation even though additional depth may be beneficial.

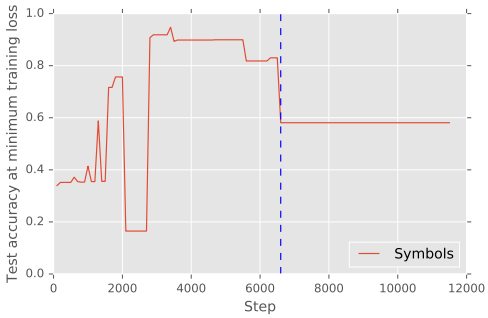


(b) This architecture started with fewer initial channels and therefore properly increases depth. This method generally generates deeper architectures.

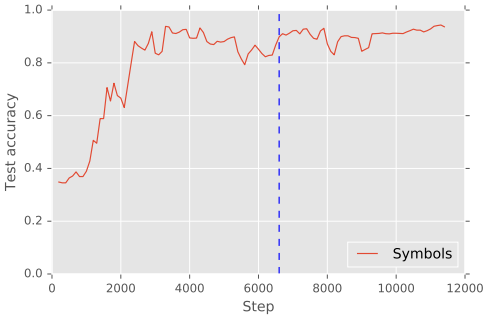
Figure 24: Visualization of failure 'depth vs channels'

The first failure case we investigate is the 'depth vs channels' issue. It affects, possibly among others, the datasets *Yoga*, *Wine*, *HandOutlines*, and *Computers*. This problem occurs whenever the loss is not reduced by using all channels in the current layer while adding a new channel would be of value. We discuss this phenomenon for the dataset *Yoga*. In Figure 24a the channels in the first convolution are not all of use, therefore gradient descent pushes the scales of a neuron below the deletion threshold. As a result, no additional layer is added and the network keeps optimizing with a single convolutional layer, restraining the possible functions f_{nn} that can be learned. This directly affects the resulting cross entropy and test accuracy, keeping them far from optimal. On the other hand Figure 24b has fewer initial channels in the first convolution, leading to subsequent expansion of the network and much better accuracies. The solution of lowering the initial number of channels may lead to deeper networks in general and is therefore far from optimal. This problem applies to only a few of the 85 datasets in the UCR archive.

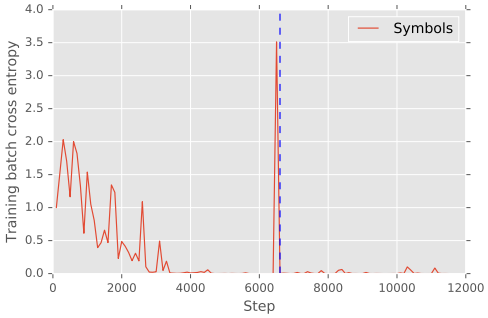
A second failure case is a short period of very low training loss that overfits perfectly to the current batch but fails to generalize. This period usually occurs right after a mutation or a scales training phase and quickly vanishes afterwards. This short loss of generalization has already been observed in subsection 5.7. If the training loss has been particularly low and fails to get even lower over all following steps, it is kept until training stops and harms the final testing accuracy as seen in Figure 25a. From Figure 25b it is obvious that the testing error is generally much better and this failure indeed only affects a short period of time. Figure 25c shows that the scales training right before the new minimal training loss had a short negative effect on the cross entropy. If enough training data is available a simple solution to this problem is the introduction of a validation set that can be used to determine the final model and when to stop training instead of the training loss. As an alternative solution, one could ensure that the lowest training loss is only updated whenever the training loss is low for several training steps. This problem affects, possibly among others, the datasets *Symbols*, *ItalyPowerDemand*, *ArrowHead*, and *FacesUCR*.



(a) From step 6,600 on the lowest training loss that overfit previously is kept



(b) The test accuracy is generally high



(c) The short period of overfitting occurred right after a scales training phase

Figure 25: Failure case: Short period of overfitting

6 Conclusion & Future work

In this thesis, we proposed DiffCNN — a CNN that not only adapts it weights using gradient descent but also its architecture. We introduced a novel regularization technique that enables the modification of the number of output channels and depth of the network by differentiation. Skip connections are inserted to maintain stable optimization. As a result, the complexity of the network can be automatically adapted and architectures need no longer be handcrafted. The remaining hyperparameters are easier to specify and have clear instructions while their default values work on a wide range of datasets. During training, we employ two different training phases that enable this joint optimization of architecture and weights. An evaluation on 85 datasets of the UCR archive showed that competitive accuracy can be achieved with this method while only requiring about three times more training time than handcrafted architectures. Additionally, by automatically adapting the network's complexity to the dataset, we outperformed existing approaches on several datasets.

The approach was only evaluated on univariate time series but could easily be expanded to multivariate time series by modeling the different series as channels of the input layer. Furthermore, in future work, the size of the convolution kernel could be learned in a similar principle by penalizing bigger kernels and pushing deactivated parts to zero. Instead of specifying the number of training steps $s_{training}$ between mutations manually, we could determine when the loss reduce has stagnated. The inserted skip connections may also be penalized to remove no longer required skip connections later in training. We outlined two shortcomings of the current approach that could be solved for even better benchmark results: Firstly, the depth of the architecture may not expand sufficiently if a layer's number of channels is not growing. Secondly, short periods after mutations and scale optimization can lead to temporary overfitting that needs to be avoided for the final solution.

A Supplementary material for experiments

Table 2: An overview of all hyperparameters and how to set their values. Parameters that are dataset sensitive are marked with stars.

Parameter	Purpose	Instructions
λ	Learning rate for the selected optimizer.	For the UCR datasets we optimize using Adam with a learning rate of $\lambda = 10^{-3}$. It also introduces parameters $\beta_1 = 0.9$, $\beta_2 = 0.999$, and $\epsilon = 10^{-8}$. The optimizer's default [29] is usually a good choice.
n_{batch}	Mini batch gradient descent uses a batch of samples for every training step. The number of samples is specified by n_{batch}	This value can be raised the more data and computational power is available, but has little effect on final testing accuracy. Common values are $n_{batch} \in \{16, 32, 64\}$. For the UCR datasets we use $n_{batch} = 16$.
$n_{stagnant}$	This optional parameter only reduces training time. The number of steps that is maximally trained with no improvement in training loss.	For the UCR datasets we use $n_{stagnant} = 5000$.
★ n_{epochs}	At some point training needs to be terminated. This may either occur through $n_{stagnant}$ or when n_{epochs} epochs have passed.	For the UCR datasets we use $n_{epochs} = 2000$ from [6].
★ K_i	Kernel size of convolutional layer i	This parameter can not be accessed by gradient descent yet. Set it to a value that is a reasonable size of the patterns to expect in the input time series. The same value can be used for all layers. For the UCR datasets we use $K_i = 16$.
DiffCNN parameter		
t_{del}	Below this threshold neurons or channels may be deleted.	Use default of $t_{del} = 5 \cdot 10^{-2}$ as defined in [9].

Table 2: An overview of all hyperparameters and how to set their values. Parameters that are dataset sensitive are marked with stars.

Parameter	Purpose	Instructions
$t_{inactive}$	The number of neurons or channels that should be inactive to allow gradient descent to reactivate them if more complexity is needed.	Use default of $t_{inactive} = 1$ as defined in [9].
$O_{initial}$	When a new layer is created, initialize with $O_{initial}$ channels.	Should be set high enough to be able to reduce the loss at all, but low enough to prevent wasting many resources until the next mutation. In this thesis $O_{initial} = 64$ is used.
O_{add}	Whenever less than $t_{inactive}$ output channels in a layer are inactive, add O_{add} many new channels.	Can be any integer $O_{add} > 0$, bigger values may lead to faster convergence to the right number of channels and shallower networks. In this thesis $O_{add} = 64$ is used.
★ α	Factor that weights the penalty in the loss function.	In general, a very small value is sufficient. Too big values lead to too much regularization. Should be about four orders of magnitude smaller than the loss at the beginning of training. In our experiments we use $\alpha = 10^{-4}$.
★ $s_{training}$	The number of training steps before the next mutation is applied.	Should be set so that the loss change saturates until the next mutation. In our experiments we use $s_{training} = 10^3$.
r_{scales}	The fraction of steps $s_{training}$ to only update the scales γ.	For the UCR datasets we use $r_{scales} = 0.1$.
λ_{scales}	Learning rate for only training the scales γ.	For the UCR datasets we use $\lambda_{scales} = 50 \cdot \lambda = 5 \cdot 10^{-2}$.

Table 3: DiffCNN and reproduced FCN testing error on all 85 UCR datasets

Error rate on dataset	Reproduced FCN	DiffCNN	Retrained DiffCNN
50words	0.347	0.301	**0.211**
Adiac	**0.169**	0.217	0.192
ArrowHead	**0.171**	0.606	0.194
Beef	0.367	**0.267**	0.3
BeetleFly	0.5	0.2	**0.150**
BirdChicken	**0.050**	0.2	0.15
Car	0.15	0.1	**0.067**
CBF	0.007	0.001	**0.000**
ChlorineCon	**0.231**	0.372	0.273
CinCECGTorso	0.176	**0.161**	0.234
Coffee	**0.000**	0	0
Computers	**0.180**	0.32	0.332
CricketX	0.197	0.21	**0.172**
CricketY	0.208	**0.197**	0.197
CricketZ	0.205	0.2	**0.149**
DiatomSizeR	0.699	0.699	**0.425**
DistalPhalanxOutlineAgeGroup	**0.190**	0.252	0.23
DistalPhalanxOutlineCorrect	**0.197**	0.198	0.198
DistalPhalanxTW	**0.220**	0.248	0.245
Earthquakes	**0.217**	0.233	0.242
ECG200	0.12	0.12	**0.110**
ECG5000	**0.065**	0.066	0.065
ECGFiveDays	0.014	**0.001**	0.001
ElectricDevices	0.288	0.379	**0.287**
FaceAll	**0.076**	0.148	0.144
FaceFour	0.057	**0.045**	0.045
FacesUCR	0.053	0.169	**0.035**
fish	0.04	0.023	**0.017**
FordA	0.101	0.078	**0.064**
FordB	0.12	**0.078**	0.104
GunPoint	**0.000**	0.007	0
Ham	0.333	**0.286**	0.476
HandOutlines	**0.210**	0.362	0.362
Haptics	0.536	0.519	**0.468**
Herring	**0.328**	0.422	0.375
InlineSkate	**0.604**	0.636	0.607
InsectWingbeatSound	0.606	0.432	**0.403**
ItalyPower	0.042	0.278	**0.037**
LargeKitchenAppliances	**0.099**	0.115	0.104
Lightning2	0.246	0.246	**0.197**
Lightning7	**0.151**	0.151	0.164
MALLAT	0.044	0.023	**0.021**
Meat	0.2	**0.017**	0.083
MedicalImages	0.246	0.238	**0.220**

Table 3: DiffCNN and reproduced FCN testing error on all 85 UCR datasets

Error rate on dataset	Reproduced FCN	DiffCNN	Retrained DiffCNN
MiddlePhalanxOutlineAgeGroup	0.3	**0.280**	0.32
MiddlePhalanxOutlineCorrect	**0.237**	0.243	0.278
MiddlePhalanxTW	**0.414**	0.416	0.424
MoteStrain	**0.063**	0.146	0.109
NonInvThorax1	0.103	0.223	**0.050**
NonInvThorax2	**0.047**	0.14	0.112
OliveOil	0.3	0.6	**0.200**
OSULeaf	**0.017**	0.05	0.054
PhalangesOutlinesCorrect	0.51	0.205	**0.193**
Phoneme	0.676	0.682	**0.669**
Plane	**0.000**	0	0
ProximalPhalanxOutlineAgeGroup	0.176	**0.127**	0.185
ProximalPhalanxOutlineCorrect	0.103	0.158	**0.100**
ProximalPhalanxTW	**0.197**	0.252	0.255
RefrigerationDevices	0.515	0.533	**0.440**
ScreenType	**0.360**	0.416	0.448
ShapeletSim	0.35	0.456	**0.261**
ShapesAll	0.1	0.113	**0.080**
SmallKitchenAppliances	0.299	0.243	**0.240**
SonyAIBORobot	0.03	0.023	**0.018**
SonyAIBORobotII	**0.027**	0.048	0.034
StarLightCurves	0.042	0.032	**0.028**
Strawberry	**0.091**	0.357	0.157
SwedishLeaf	**0.035**	0.048	0.053
Symbols	**0.051**	0.419	0.062
SyntheticControl	0.02	**0.003**	0.003
ToeSegmentation1	0.044	0.048	**0.039**
ToeSegmentation2	0.077	**0.062**	0.069
Trace	**0.000**	0	0
TwoLeadECG	0.001	**0.000**	0.001
TwoPatterns	0.167	**0.000**	0
UWaveGestureLibraryAll	0.185	0.154	**0.104**
UWaveX	0.248	0.238	**0.209**
UWaveY	0.389	0.333	**0.281**
UWaveZ	**0.275**	0.276	0.276
wafer	0.003	0.007	**0.002**
Wine	**0.296**	0.5	0.5
WordSynonyms	0.445	0.342	**0.315**
Worms	**0.315**	0.387	0.365
WormsTwoClass	0.343	**0.249**	0.249
yoga	**0.191**	0.416	0.443
Wins	**50**	35	—
Wins Retrained	40	—	**45**
MPCE	0.055	0.063	**0.054**

Table 4: Testing error of the DiffCNN and its competitors as given in [6] on 44 UCR datasets

Error rate on dataset	DTW	COTE	MCNN	BOSSVS	PROP	BOSS	SE1	TSBF	MLP	FCN	ResNet	DiffCNN
Adiac	0.396	0.233	0.231	0.302	0.353	0.22	0.373	0.245	0.248	**0.143**	0.174	0.192
Beef	0.367	**0.133**	0.367	0.267	0.367	0.2	0.133	0.287	0.167	0.25	0.233	0.3
CBF	0.003	0.001	0.002	0.001	0.002	**0.000**	0.01	0.009	0.14	0	0.006	0
ChlorineCon	0.352	0.314	0.203	0.345	0.36	0.34	0.312	0.336	**0.128**	0.157	0.172	0.273
CinCECGTorso	0.349	0.064	0.058	0.13	0.062	0.125	**0.021**	0.262	0.158	0.187	0.229	0.234
Coffee	**0.000**	0	0.036	0.036	0	0	0	0.004	0	0	0	0
CricketX	0.246	**0.154**	0.182	0.346	0.203	0.259	0.297	0.278	0.431	0.185	0.179	0.172
CricketY	0.256	0.167	**0.154**	0.328	0.156	0.208	0.326	0.259	0.405	0.208	0.195	0.197
CricketZ	0.246	**0.128**	0.142	0.313	0.156	0.246	0.277	0.263	0.408	0.187	0.187	0.149
DiatomSizeR	0.033	0.082	**0.023**	0.036	0.059	0.046	0.069	0.126	0.036	0.07	0.069	0.425
ECGFiveDays	0.232	**0.000**	0	0	0.178	0	0.055	0.183	0.03	0.015	0.045	0.001
FaceAll	0.192	0.105	0.235	0.241	0.152	0.21	0.247	0.234	0.115	**0.071**	0.166	0.144
FaceFour	0.17	0.091	**0.000**	0.034	0.091	0	0.034	0.051	0.17	0.068	0.068	0.045
FacesUCR	0.095	0.057	0.063	0.103	0.063	0.042	0.079	0.09	0.185	0.052	0.042	**0.035**
50words	0.31	0.191	0.19	0.367	**0.180**	0.301	0.288	0.209	0.288	0.321	0.273	0.211
fish	0.177	0.029	0.051	0.017	0.034	**0.011**	0.057	0.08	0.126	0.029	0.011	0.017
GunPoint	0.093	0.007	**0.000**	0	0.007	0	0.06	0.011	0.067	0	0.007	0
Haptics	0.623	0.488	0.53	0.584	0.584	0.536	0.607	0.488	0.539	**0.449**	0.495	0.468
InlineSkate	0.616	0.551	0.618	0.573	0.567	**0.511**	0.653	0.603	0.649	0.589	0.635	0.607
ItalyPower	0.05	0.036	**0.030**	0.086	0.039	0.053	0.053	0.096	0.034	0.03	0.04	0.037
Lightning2	0.131	0.164	0.164	0.262	0.115	0.148	**0.098**	0.257	0.279	0.197	0.246	0.197
Lightning7	0.274	0.247	0.219	0.288	0.233	0.342	0.274	0.262	0.356	**0.137**	0.164	0.164
MALLAT	0.066	0.036	0.057	0.064	0.05	0.058	0.092	0.037	0.064	**0.020**	0.021	0.021
MedicalImages	0.263	0.258	0.26	0.474	0.245	0.288	0.305	0.269	0.271	**0.208**	0.228	0.22
MoteStrain	0.165	0.085	0.079	0.115	0.114	0.073	0.113	0.135	0.131	**0.050**	0.105	0.109
NonInvThorax1	0.21	0.093	0.064	0.169	0.178	0.161	0.174	0.138	0.058	**0.039**	0.052	0.05
NonInvThorax2	0.135	0.073	0.06	0.118	0.112	0.101	0.118	0.13	0.057	**0.045**	0.049	0.112
OliveOil	0.167	0.1	0.133	0.133	0.133	0.1	0.133	**0.090**	0.6	0.167	0.133	0.2
OSULeaf	0.409	0.145	0.271	0.074	0.194	**0.012**	0.273	0.329	0.43	0.012	0.021	0.054
SonyAIBORobot	0.275	0.146	0.23	0.265	0.293	0.321	0.238	0.175	0.273	0.032	**0.015**	0.018
SonyAIBORobotII	0.169	0.076	0.07	0.188	0.124	0.098	0.066	0.196	0.161	0.038	0.038	**0.034**
StarLightCurves	0.093	0.031	0.023	0.096	0.079	**0.021**	0.093	0.022	0.043	0.033	0.029	0.028
SwedishLeaf	0.208	0.046	0.066	0.141	0.085	0.072	0.12	0.075	0.107	**0.034**	0.042	0.053
Symbols	0.05	0.046	0.049	**0.029**	0.049	0.032	0.083	0.034	0.147	0.038	0.128	0.062
SyntheticControl	0.007	**0.000**	0.003	0.04	0.01	0.03	0.033	0.008	0.05	0.01	0	0.003
Trace	**0.000**	0.01	0	0	0.01	0	0.05	0.02	0.18	0	0	0
TwoLeadECG	**0.000**	0.015	0.001	0.015	0	0.004	0.029	0.001	0.147	0	0	0.001
TwoPatterns	0.096	**0.000**	0.002	0.001	0.067	0.016	0.048	0.046	0.114	0.103	0	0
UWaveGestureLibraryAll	0.272	0.196	0.18	0.27	0.199	0.241	0.248	0.164	0.232	0.246	0.213	**0.104**
UWaveX	0.366	0.267	0.268	0.364	0.283	0.313	0.322	0.249	0.297	0.275	0.332	**0.209**
UWaveY	0.342	0.265	0.232	0.336	0.29	0.312	0.346	**0.217**	0.295	0.271	0.245	0.281
wafer	0.02	**0.001**	0.002	0.001	0.003	0.001	0.002	0.004	0.004	0.003	0.003	0.002
WordSynonyms	0.351	0.266	0.276	0.439	**0.226**	0.345	0.357	0.302	0.406	0.42	0.368	0.315
yoga	0.164	0.113	0.112	0.169	0.121	**0.081**	0.159	0.149	0.145	0.155	0.142	0.443
Wins	3	7	5	1	2	6	2	2	1	**10**	1	4
Average arithmetic ranking	9.273	**4.42**	4.761	8.295	6.523	5.534	8.477	7.409	8.807	4.727	5.034	4.739
Average geometric ranking	8.625	3.832	4.059	7.29	5.734	4.475	7.565	6.42	7.905	**3.514**	4.254	3.878
MPCE	0.039	0.023	0.024	0.033	0.03	0.026	0.03	0.033	0.04	**0.021**	0.023	0.028

References

[1] S. Hochreiter and J. Schmidhuber, "Long short-term memory," *Neural Computation*, vol. 9, pp. 1735–1780, Nov. 1997.

[2] Y. LeCun, Y. Bengio, *et al.*, "Convolutional networks for images, speech, and time series," *The handbook of brain theory and neural networks*, vol. 3361, no. 10, p. 1995, 1995.

[3] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pp. 770–778, 2016.

[4] R. Collobert, C. Puhrsch, and G. Synnaeve, "Wav2letter: an end-to-end convnet-based speech recognition system," *CoRR*, vol. abs/1609.03193, 2016.

[5] Z. Cui, W. Chen, and Y. Chen, "Multi-scale convolutional neural networks for time series classification," *CoRR*, vol. abs/1603.06995, 2016.

[6] Z. Wang, W. Yan, and T. Oates, "Time series classification from scratch with deep neural networks: A strong baseline," *CoRR*, vol. abs/1611.06455, 2016.

[7] E. Real, S. Moore, A. Selle, S. Saxena, Y. L. Suematsu, Q. V. Le, and A. Kurakin, "Large-scale evolution of image classifiers," *CoRR*, vol. abs/1703.01041, 2017.

[8] B. Zoph and Q. V. Le, "Neural architecture search with reinforcement learning," *CoRR*, vol. abs/1611.01578, 2016.

[9] T. Miconi, "Neural networks with differentiable structure," *CoRR*, vol. abs/1606.06216, 2016.

[10] Y. Chen, E. Keogh, B. Hu, N. Begum, A. Bagnall, A. Mueen, and G. Batista, "The ucr time series classification archive," July 2015.

[11] E. Keogh and C. A. Ratanamahatana, "Exact indexing of dynamic time warping," *Knowledge and information systems*, vol. 7, no. 3, pp. 358–386, 2005.

[12] M. G. Baydogan, G. Runger, and E. Tuv, "A bag-of-features framework to classify time series," *IEEE Transactions on pattern analysis and machine intelligence*, vol. 35, no. 11, pp. 2796–2802, 2013.

[13] P. Schäfer, "The boss is concerned with time series classification in the presence

of noise," *Data Mining and Knowledge Discovery*, vol. 29, no. 6, pp. 1505–1530, 2015.

[14] P. Schäfer, "Scalable time series classification," *Data Mining and Knowledge Discovery*, vol. 30, no. 5, pp. 1273–1298, 2016.

[15] J. Lines and A. Bagnall, "Time series classification with ensembles of elastic distance measures," *Data Mining and Knowledge Discovery*, vol. 29, no. 3, pp. 565–592, 2015.

[16] A. Bagnall, J. Lines, J. Hills, and A. Bostrom, "Time-series classification with cote: the collective of transformation-based ensembles," *IEEE Transactions on Knowledge and Data Engineering*, vol. 27, no. 9, pp. 2522–2535, 2015.

[17] Y. L. Cun, J. S. Denker, and S. A. Solla, "Optimal brain damage," in *Advances in Neural Information Processing Systems 2* (D. S. Touretzky, ed.), pp. 598–605, San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1990.

[18] S. Han, J. Pool, J. Tran, and W. J. Dally, "Learning both weights and connections for efficient neural networks," in *Proceedings of the 28th International Conference on Neural Information Processing Systems*, NIPS'15, (Cambridge, MA, USA), pp. 1135–1143, MIT Press, 2015.

[19] H. Kitano, "Designing neural networks using genetic algorithms with graph generation system," *Complex systems*, vol. 4, no. 4, pp. 461–476, 1990.

[20] N. Y. Nikolaev and H. Iba, "Learning polynomial feedforward neural networks by genetic programming and backpropagation," *IEEE Transactions on Neural Networks*, vol. 14, no. 2, pp. 337–350, 2003.

[21] P. P. Palmes, T. Hayasaka, and S. Usui, "Mutation-based genetic neural network," *IEEE Transactions on Neural Networks*, vol. 16, no. 3, pp. 587–600, 2005.

[22] X. Glorot, A. Bordes, and Y. Bengio, "Deep sparse rectifier neural networks," in *Proceedings of the Fourteenth International Conference on Artificial Intelligence and Statistics*, pp. 315–323, 2011.

[23] D. E. Rumelhart, G. E. Hinton, R. J. Williams, *et al.*, "Learning representations by back-propagating errors," *Cognitive modeling*, vol. 5, no. 3, p. 1, 1988.

[24] I. Goodfellow, Y. Bengio, and A. Courville, *Deep Learning*. MIT Press, 2016.

[25] A. Y. Ng, "Feature selection, l 1 vs. l 2 regularization, and rotational invariance," in *Proceedings of the twenty-first international conference on Machine learning*, p. 78, ACM, 2004.

[26] M. Lin, Q. Chen, and S. Yan, "Network in network," *CoRR*, vol. abs/1312.4400, 2013.

[27] S. Ioffe and C. Szegedy, "Batch normalization: Accelerating deep network training by reducing internal covariate shift," in *Proceedings of the 32nd International Conference on Machine Learning* (F. Bach and D. Blei, eds.), vol. 37 of *Proceedings of Machine Learning Research*, (Lille, France), pp. 448–456, PMLR, July 2015.

[28] D. Weigend, "Back-propagation, weight-elimination and time series prediction," in *Proceedings 1990 Connectionist Models Summer School*, pp. 105–116, 1990.

[29] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," *CoRR*, vol. abs/1412.6980, 2014.

[30] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin, S. Ghemawat, I. Goodfellow, A. Harp, G. Irving, M. Isard, Y. Jia, R. Jozefowicz, L. Kaiser, M. Kudlur, J. Levenberg, D. Mané, R. Monga, S. Moore, D. Murray, C. Olah, M. Schuster, J. Shlens, B. Steiner, I. Sutskever, K. Talwar, P. Tucker, V. Vanhoucke, V. Vasudevan, F. Viégas, O. Vinyals, P. Warden, M. Wattenberg, M. Wicke, Y. Yu, and X. Zheng, "TensorFlow: Large-scale machine learning on heterogeneous systems," 2015. Software available from tensorflow.org.

[31] X. Glorot and Y. Bengio, "Understanding the difficulty of training deep feedforward neural networks," in *Proceedings of the Thirteenth International Conference on Artificial Intelligence and Statistics*, pp. 249–256, 2010.

[32] K. He and J. Sun, "Convolutional neural networks at constrained time cost," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pp. 5353–5360, 2015.

[33] M. Banko and E. Brill, "Scaling to very very large corpora for natural language disambiguation," in *Proceedings of the 39th Annual Meeting on Association for Computational Linguistics*, ACL '01, (Stroudsburg, PA, USA), pp. 26–33, Association for Computational Linguistics, 2001.

German abstract

In nur wenigen Jahren hat Deep Learning den Bereich des maschinellen Lernens revolutioniert. Ein sehr wichtiger Bereich ist die Zeitreihenklassifikation. Zeitreihen entstehen in vielen realen Szenarien, beispielsweise als industrielle oder medizinische Sensordaten. Der Erfolg des Deep Learning kann hauptsächlich auf das Trainieren durch Rückwärtspropagierung zurück geführt werden. Trotz dieses Erfolges kann die Rückwärtspropagierung nur für das Trainieren der Gewichte verwendet werden. Da das Konfigurieren der Architektur allerdings nicht differenzierbar ist, ist hier eine Anwendung nicht möglich. Dies hat zur Folge, dass der aktuelle Stand der Wissenschaft häufig auf manuell erstellte Architekturen zurückgreift. Diese Abschlussarbeit zielt darauf ab, automatisch neue Architekturen für die Zeitreihenklassifikation bei geringem Zeitaufwand zu generieren. Moderne Lösungen der algorithmischen Erstellung solcher Architekturen verwenden meist evolutionäre oder Methoden des bestärkenden Lernens. Diese sind in ihrer Verwendung deutlich rechenintensiver. Wir lösen das Problem, indem wir eine neue Regularisierungstechnik für neuronale Netzwerke basierend auf Faltung (CNNs) einführen, die es ermöglicht, das Trainieren der Architektur und der Gewichte des Netzwerkes gleichzeitig durchzuführen. Wir führen sowohl Netzwerkverbindungen, die Operationen überspringen ein, als auch ein neuartiges zweiphasiges Training, welches eine stabile Optimierung ermöglicht. Die Arbeit wird auf den Datensätzen des UCR Archivs evaluiert und zeigt konkurrenzfähige Ergebnisse zu aktuellen Bestverfahren. Auf Datensätzen, bei denen die händisch gebaute Architektur nicht ausreichend ist, erreicht das Verfahren bessere Werte als bisher durch andere Ansätze erreicht.

Selbstständigkeitserklärung

Hiermit erkläre ich, die vorliegende Arbeit selbstständig angefertigt, nicht anderweitig zu Prüfungszwecken vorgelegt und keine anderen als die angegebenen Hilfsmittel verwendet zu haben. Sämtliche wissentlich verwendete Textausschnitte, Zitate oder Inhalte anderer Verfasser wurden ausdrücklich als solche gekennzeichnet.

Potsdam, 20. Juli 2017

Louis Kirsch