Master Thesis

An Evaluation of Layer Transferability for Temporal Convolutional Networks

Robin Middelanis

Supervision

Université de Rennes 1: Prof. Dr. Guillaume Pierre
IAV GmbH: M.Sc. Alexander Oberneyer

Berlin, August 2019

Abstract

Deep neural networks have shown remarkable results on learning a mapping between inputs and desired outputs. However, these successes are mostly limited to tasks and domains where vast amounts of (labeled) data are available. To reduce the amount of necessary training data, Transfer Learning approaches aim to leverage knowledge from already existing models. In the area of Machine Vision, using pre-trained weights from existing Convolutional Neural Networks (CNN) is common practice. While convolutional architectures like WaveNet are also being used for sequence modeling in practice, layer transferability of such Temporal Convolutional Networks (TCN) is still an open research issue. With this work, we aim to close this gap. We investigate the transferability of TCN layers in a qualitative way by successively transferring layers and evaluating the performance of the resulting models. We find two aspects affecting the transfer performance: (1) different amounts and kind of information in the source and target dataset can result in a positive or negative effect on the transfer, and (2) specificity of deep layers reduces the transfer performance. For a better understanding of the kind of knowledge that is transferred, we visualize the activation of intermediate layers in the network. Our results suggest that different neurons perform general tasks like smoothing, filtering, inverting and shifting. Finally, we measure the amount of knowledge that each layer contributes to the overall predictive power of the model and find that the low, general layers exhibit the biggest influence. Deeper layers, while more specific, contribute less to the predictive power of the model. Our experimental setup also shows to be an efficient approach for identifying the threshold in a network where layers transition from general to specific, given a source and a target dataset.

Contents

List of Figures vii

List of Tables viii

Nomenclature ix

Acknowledgements xii

1 Introduction 1
- 1.1 Problem and Research Questions . . . 1
- 1.2 Thesis Structure . . . 3

2 Foundations and Related Work 5
- 2.1 Time Series Modeling . . . 5
  - 2.1.1 Recurrent Neural Network Architectures . . . 6
  - 2.1.2 Temporal Convolutional Networks . . . 7
- 2.2 Transfer Learning . . . 12
- 2.3 Visualization of Convolutional Neural Networks . . . 15
- 2.4 Synthetic Data Generation . . . 16

3 Qualitative Evaluation of Layer Transferability 19
- 3.1 Experiment Overview . . . 19
  - 3.1.1 Main Experiments . . . 20
  - 3.1.2 Reference Experiment . . . 22
- 3.2 Training and Evaluation . . . 24
  - 3.2.1 Training Parameters . . . 24
  - 3.2.2 Training Steps . . . 24
  - 3.2.3 Evaluation and Significance of Performance Differences . . . 25
- 3.3 Experiment Results . . . 27
  - 3.3.1 Hard Feature Extraction . . . 27
  - 3.3.2 Soft Feature Extraction . . . 28
  - 3.3.3 Full Weight Initialization with partial Freeze . . . 29
  - 3.3.4 Random Reference . . . 30
  - 3.3.5 Reduction of Dataset Knowledge . . . 33
  - 3.3.6 Training Duration . . . 34
- 3.4 Discussion . . . 38

4 Knowledge Visualization **42**
4.1 Layer Activation . . . 42
4.2 Visualizing Layer Modification . . . 44
4.3 Horizontal Evaluation . . . 47
4.4 Vertical Evaluation . . . 49
4.5 Discussion . . . 50

5 Quantification of Layer Knowledge **52**
5.1 Experiment Overview . . . 52
5.2 Experiment Results . . . 54
5.3 Discussion . . . 57

6 Conclusion **58**
6.1 Results . . . 58
6.2 Future Work . . . 60

A Supplementary Material: Qualitative Evaluation of Layer Transferabilty **65**
A.1 Loss Plots . . . 65
A.1.1 Transfer from Dataset B to Dataset A - Reduced Datasets . . . 66
A.1.2 Transfer from Dataset A to Dataset B - Complete Datasets . . . 68
A.1.3 Transfer from Dataset A to Dataset B - Reduced Datasets . . . 70
A.2 Probability Plots . . . 73
A.2.1 Transfer from Dataset B to Dataset A . . . 73
A.2.2 Transfer from Dataset A to Dataset B . . . 78

B Supplementary Material: Knowledge Visualization **82**
B.1 Activation Plots . . . 82
B.1.1 Dataset A . . . 83
B.1.2 Dataset B . . . 92
B.2 Horizontal Evaluation . . . 101
B.3 Vertical Evaluation . . . 103

List of Figures

2.1 Recurrent Neural Network with one hidden layer 6
2.2 Unfolded Recurrent Neural Network 6
2.3 Architecture of the Long Short-Term Memory 7
2.4 Dilated convolution with kernel size 2 and exponential dilation growth . . . 8
2.5 Residual Block in the TCN architecture 10

3.1 Visualization of the qualitative evaluation setup for transfer level $l = 5$ . . . 23
3.2 Loss plot for the Hard Feature Extraction experiment from dataset B to dataset A 28
3.3 Loss plot for the Soft Feature Extraction experiment from dataset B to dataset A 30
3.4 Loss plot for the Soft Feature Extraction experiment from dataset A to dataset B 31
3.5 Loss plot for the Full Weight Initialization with partial Freeze from dataset B to dataset A 32
3.6 Loss plot for the random reference experiment with target dataset A 33
3.7 Loss plot for the Hard Feature Extraction experiment from dataset B to dataset A with reduced datasets 34
3.8 Duration plots for the three main experiments 37
3.9 Loss plot of all main experiments and the reference experiment. 39
3.10 Loss plot of the Hard Feature Extraction, Soft Feature Extraction and random reference experiment transfer models with marked areas 41

4.1 Location of hooks in the TCN for the visualization of layer activations . . . 43
4.2 Activation of the maximally activated neuron in each layer for different input signals 46
4.3 Influence of residual blocks on the total activation of the network trained on dataset B 48
4.4 Activation distribution of all fifty channels of the final network output . . . 50

5.1 Visualization of the quantitative evaluation setup for evaluation level $l = 5$. 53
5.2 Loss plots of the quantitative experiment for the evaluation on dataset A as target dataset 55
5.3 Loss plots of the quantitative experiment for the evaluation on dataset B as target dataset 56

A.1 Loss plot for the Soft Feature Extraction experiment from dataset B to dataset A with reduced datasets . . . 66
A.2 Loss plot for the Full Weight Initialization with partial Freeze experiment from dataset B to dataset A with reduced datasets . . . 67
A.3 Loss plot for the Hard Feature Extraction experiment from dataset A to dataset B with complete datasets . . . 68
A.4 Loss plot for the Full Weight Initialization with partial Freeze experiment from dataset A to dataset B with complete datasets . . . 69
A.5 Loss plot for the Hard Feature Extraction experiment from dataset A to dataset B with reduced datasets . . . 70
A.6 Loss plot for the Soft Feature Extraction experiment from dataset A to dataset B with reduced datasets . . . 71
A.7 Loss plot for the Full Weight Initialization with partial Freeze experiment from dataset A to dataset B with reduced datasets . . . 72
A.8 Probability plot for the Hard Feature Extraction experiment from dataset B to dataset A with complete datasets . . . 74
A.9 Probability plot for the Soft Feature Extraction experiment from dataset B to dataset A with complete datasets . . . 75
A.10 Probability plot for the Full Weight Initialization with partial Freeze experiment from dataset B to dataset A with complete datasets . . . 76
A.11 Probability plot for the random reference experiment from random weights to dataset A with complete datasets . . . 77
A.12 Probability plot for the Hard Feature Extraction experiment from dataset A to dataset B with complete datasets . . . 78
A.13 Probability plot for the Soft Feature Extraction experiment from dataset A to dataset B with complete datasets . . . 79
A.14 Probability plot for the Full Weight Initialization with partial Freeze experiment from dataset A to dataset B with complete datasets . . . 80
A.15 Probability plot for the random reference experiment from random weights to dataset B with complete datasets . . . 81

B.1 Modification activation plots of a model trained on dataset A to a constant input signal . . . 84
B.2 Modification activation plots of a model trained on dataset A to a dirac function input signal . . . 85
B.3 Modification activation plots of a model trained on dataset A to a linear input signal . . . 86
B.4 Modification activation plots of a model trained on dataset A to a sawtooth input signal . . . 87
B.5 Modification activation plots of a model trained on dataset A to a sine wave input signal . . . 88

B.6 Modification activation plots of a model trained on dataset A to a square wave input signal . . . 89
B.7 Modification activation plots of a model trained on dataset A to a random test sequence input . . . 90
B.8 Modification activation plots of a model trained on dataset A to a triangular wave input signal . . . 91
B.9 Modification activation plots of a model trained on dataset B to a constant input signal . . . 93
B.10 Modification activation plots of a model trained on dataset B to a dirac function input signal . . . 94
B.11 Modification activation plots of a model trained on dataset B to a linear input signal . . . 95
B.12 Modification activation plots of a model trained on dataset B to a sawtooth input signal . . . 96
B.13 Modification activation plots of a model trained on dataset B to a sine wave input signal . . . 97
B.14 Modification activation plots of a model trained on dataset B to a square wave input signal . . . 98
B.15 Modification activation plots of a model trained on dataset B to a random test sequence input . . . 99
B.16 Modification activation plots of a model trained on dataset B to a triangular wave input signal . . . 100
B.17 Influence of residual blocks on the total activation of the network trained on dataset A . . . 101
B.18 Activation distribution of all fifty channels of the final network output . . . 103

List of Tables

3.1 Experiment table for the qualitative transferability evaluation 21
3.2 Training durations of the three main experiments and the reference experiment 25

4.1 Layer influence for different input signals of a TCN trained on dataset B . . 48
4.2 Number and percentage of dead modifications in each residual block for two models trained on dataset A and dataset B 50

B.1 Layer influence for different input signals of a TCN trained on dataset A . . 102

Nomenclature

Symbols

A	dataset A
a	activation level
$\mathbf{a}$	activation vector
amp	amplitude
α	type I error probability
B	dataset B
$\mathcal{D}$	domain
d	dilation factor
Δ	difference of sample means
$\mathbb{E}$	expected value
F	receptive field size
f	prediction function
$freq$	frequency
H	hypothesis
I	number of neurons in a layer
i	residual block influence on the output activation
k	kernel size
κ	relative layer knowledge
L	number of layers
λ	model prediction loss
len	signal length
$\mathcal{M}$	instance of a TCN model
mod	residual block modification level
$\mathbf{mod}$	residual block modification vector
μ	mean value
N	number of residual blocks
ν	degrees of freedom
out	output activation level of a residual block

$\mathbf{out}$	output activation vector of a residual block
P	marginal probability distribution over a feature space
T	sequence length
$\mathcal{T}$	task
X	set of samples
$\mathcal{X}$	feature space
x	input sequence
$\mathbf{x}$	training sequence in X
Y	set of true target values
$\mathcal{Y}$	target value space
y	target sequence
$\hat{y}$	predicted sequence

Subscripts

A	references dataset A
$A \to B$	references a transfer from dataset A to dataset B
B	references dataset B
$B \to A$	references a transfer from dataset B to dataset A
i	neuron index
l	layer index
max	maximum of the variable with regards to some subscript index
n	residual block index
S	references the source domain
$S \to T$	references a transfer model
s	superposition index for synthetic data generation
T	references the target domain
$T \to T$	references a selffer model
t	time step t
$test$	reference the test set of a dataset
$train$	reference the training set of a dataset
$\cdot$	aggregate over a subscript index

Superscripts

i	neuron index
in	references the input to the network
n	residual block index
out	references the output of the network
$\cdot$	aggregate over a superscript index
$*$	indicates a reduced base model
$'$	context specific alteration to a variable
$\bar{}$	average of a variable

Acronyms and Abbreviations

ANN	Artificial Neural Network
CNN	Convolutional Neural Network
GRU	Gated Recurrent Unit
GPM	Gaussian Process Model
IAV	Ingenieurgesellschaft Auto und Verkehr GmbH
LSTM	Long Short-Term Memory
MLP	Multi Layer Perceptron
MSE	Mean Squared Error
NLP	Natural Language Processing
ReLU	Rectified Linear Unit
RNN	Recurrent Neural Network
SVM	Support Vector Machine
TCN	Temporal Convolutional Network
TDNN	Time-Delay Neural Network

Acknowledgements

I would like to express my gratitude towards my supervisor at IAV, Alexander Oberneyer, who made this work possible in the first place. His friendly guidance throughout all stages of this work, his expert advice and his constructive feedback have proven invaluable during the entire time that I have worked on this thesis.
My sincere thanks also go to my supervisor at Université de Rennes 1, Professor Guillaume Pierre, who - despite the distance between Rennes and Berlin - provided feedback to any of my questions during the thesis period.
This thesis was written during my time at IAV GmbH in Berlin. I am very grateful that I had the opportunity to work in a department that I perceived as an inspiring, friendly and intellectually challenging environment. Special thanks are due to Dr. Wolf Baumann, Dr. Simon Steinberg and Dr. Michael Hegmann for supporting me with their professional expertise and valuable feedback to my work.

Berlin, August 15th, 2019

Robin Middelanis

Chapter 1

Introduction

In the past, deep neural networks have shown remarkable results on learning a mapping between inputs and desired outputs. However, these successes are mostly limited to tasks and domains where vast amounts of (labeled) data are available and they often come along with extensive training times. Typically, a new model is trained from scratch for every new task. As opposed to this, Transfer Learning approaches aim to leverage knowledge from already existing models, thus reducing both the amount of necessary training data and the required training time. In the area of Machine Vision, using pre-trained weights from existing Convolutional Neural Networks (CNN) is common practice. While convolutional architectures like WaveNet are being used for sequence modeling in practice, layer transferability of such Temporal Convolutional Networks (TCN) is still an open research issue. With this work, we aim to close this gap.
In the following, we introduce the use case at IAV for time series modeling in the domain of combustion engine calibration and the motivation to apply Transfer Learning in this area. Based on this motivation, we derive the research questions that define the scope of this work, followed by an overview on the structure of this thesis.

1.1 Problem and Research Questions

The behaviour of a running combustion engine depends heavily on a number of external parameters like load requirement, rotation speed, boost pressure, injection point and other variables that define the state of the engine. Typically, the number of such parameters ranges between 12 to 15 independent variables. When developing a vehicle, car manufacturers want to make sure that the engine behaves in a defined way, either due to legal requirements or as a product decision. For example, central requirements by car manufacturers for combustion engines concern emissions of the engine under different driving conditions. To make sure that the state space of the engine is in accordance with these requirements, correct adjustment of the external engine parameters is necessary. The development process of finding the correct parameters for different driving situations is called *engine calibration*. A frequent calibration scenario at IAV is to find the parameters for some engine such that its emissions fulfill the customer's and legal requirements. The engine emissions in this case can be regarded as a dependent variable that is influenced

by independent external variables, i.e. named parameters.
The naive approach to calibrate an engine would be to physically test all engine states with regards to the dependent variable of interest and to find this way the optimal set of parameters. However, with only 12 independent variables, this would lead to a total number of 4096 states only to test corner cases. The resulting set of points in the state space would still be relatively sparse. Additional complexity is added through engine dynamics, meaning that not only stationary states in the defined dimensions are of interest but also non-stationary ones. In other words, transitions from one state to another need to be considered. Due to the resulting variety of necessary experiments, the naive approach is practically unfeasible.
Instead, the typical approach for calibrating an engine is to only physically test a subset of all possible engine states. This subset for testing is defined through an appropriate *Design of Experiments* (DoE) that defines a statistically meaningful and practically feasible set of physical experiments to conduct. The goal is to run a limited number of physical experiments that cover the engine's characteristics with an influence on the dependent variable of interest. With the data from these experiments, a model for the dependent variable is generated which can then be used to simulate points in the state space of the engine. The result is a more dense set of points than through physical experiments, allowing to find a set of optimal calibration parameters.
At IAV, typical modeling approaches are Gaussian Process Models (GPM) and Volterra series. IAV has also experimented with Artificial Neural Networks (ANN) and in particular Convolutional Neural Networks (CNN) to generate models. Most recently, WaveNet [20] as introduced by van den Oord et al. were used to model engine test bench series. These models use a one-dimensional convolution with increased receptive field size through dilation of the filter kernels. The achieved performance promises a good alternative for the usually used GPMs and Volterra series.
Despite promising performance results with WaveNet, there are still some caveats with regards to this approach. One main disadvantage of WaveNet is the longer training time in comparison to the traditional methods. At the same time, training a Wavenet model with comparative performance to GPMs or Volterra series requres similar amounts of training data, hence costly and time-consuming test bench experiments are not reduced. To constitute a competitive approach with the traditional approaches, a neural network architecture would need to (a) train faster than current approaches or (b) reduce the amount of necessary training data. Both aspects are investigated in this work with regards to the use of *Transfer Learning*.
For Machine Vision networks, especially CNNs, Transfer Learning has been heavily used, mainly to reduce the amount of necessary training data [21]. The idea is to transfer knowledge from an existing network to a new model under the hypothesis that some of the knowledge in the source network is general and therefore beneficial for the target task. For example, Machine Vision models have been shown to learn general features like edge filters in lower layers of the network [16] that are well transferable. While both the degree

of generality and the transferability of knowledge in the layers of Machine Vision networks have been researched in the past [24], to our knowledge these questions have not been investigated yet for time series models like WaveNet or related architectures, so-called *Temporal Convolutional Networks* (TCN). With the research presented in this work, we aim to close this gap. Instead of working with the Wavenet architecture, we choose to investigate a simpler but related architecture introduced by Bai et al. [3]. Whenever we refer to TCN architectures in the remainder of this work, this is the particular architecture in question. Based on the established use of Transfer Learning for CNNs in the domain of Machine Vision and the knowledge that TCNs for time series modeling are a special case of convolutional networks, we derive the following three research questions:

1. Can the knowledge learned in a TCN be transferred to other tasks and domains?
2. If parts of a TCN model are transferable, what kind of knowledge is encoded in the transferable features?
3. How much do the different layers in a TCN contribute to the overall predictive power of the model?

1.2 Thesis Structure

The content of this thesis follows the three introduced research questions, dedicating one chapter to each of the questions. The document is structured in six chapters. Following this introductory chapter, in **chapter 2** we give an overview of the foundations that our research is based on, as well as relevant related literature. We introduce the purpose of time series modeling and present existing approaches with a focus on neural network architectures for time series modeling, including a detailed description of the TCN architecture. We also introduce the concept of Transfer Learning and common approaches thereof, followed by an overview of visualization techniques that have been developed to better understand the kind of knowledge that CNN layers contain. Finally, we give insight into how we generate the synthetic dataset that is used throughout most experiments of this work.
Chapters 3 through 5 are each dedicated to one of the three research questions. **Chapter 3** focuses on the first research question with the goal to find out if the knowledge inside a TCN is transferable to other tasks and domains. After an introduction to the respective experiments, we provide details on the training and evaluation process before separately presenting the results of each of the experiments. Besides looking at the predictive performance of models throughout the experiments, we also investigate the training duration that we observe during the experiments and how it is affected by the knowledge transfer. Finally, we compare the results of the different experiments and discuss the findings.
In **chapter 4**, we present our work on the second research question regarding the kind of knowledge that TCN features contain. To understand what TCNs learn, we analyze the activation of individual layers and neurons for different model inputs. We visualize

intermediate activations of neurons and quantify the degree of activation throughout the network.

Chapter 5 addresses the third research question on the quantification of layer knowledge. We analyze how much each layer in the network contributes to the overall predictive power of the model. An introduction to the approach of knowledge quantification is followed by a discussion of the experiment results.

In the final **chapter 6**, we conclude the work presented in this thesis and give an outlook on future research that may build on our findings.

Chapter 2

Foundations and Related Work

To answer the previously introduced research questions, we develop experiments for a qualitative analysis of layer transferability, visualization of layer knowledge and a quantitative transferability evaluation. The experiments are based on the theoretical foundations of time series modeling with neural nets and approaches layer visualization in CNNs. In this chapter, we give an introduction to the theoretical foundations and literature relevant for our research. In section 2.1, we present an overview of neural network architectures for time series modeling. The notion of Transfer Learning and common approaches are introduced along with existing evaluation techniques of layer transferability in section 2.2. To understand what kind of knowledge is learned in Convolutional Neural Network, different visualization techniques have been proposed which we describe in section 2.3. Finally, we explain in section 2.4 how we generate the data that we use throughout most of the experiments.

2.1 Time Series Modeling

The goal of time series modeling is to generate a model with the ability to describe the inherent structure of some time series. A time series is defined as a set of values over time in chronological order and can be regarded as a random variable. Time series can be univariate, i.e. they contain records of only one variable, or multivariate, i.e. more than one variable is regarded over time. If the variables are measured at every point in time, they are called continuous, otherwise discrete [1, p. 11 ff.]. To generate a model that describes the series, past observations of this time series are collected and analyzed. The generation of a time series model may serve different purposes, e.g. predicting future values, filtering, hypothesis testing or simply to provide a compact description of the series [5, p. 6 f.]. To obtain models of a series, different approaches can be found in the literature. These comprise among others stochastic methods, Support Vector Machines (SVM) and Artificial Neural Networks (ANN). Since our work analyzes a specific architecture of ANNs, we limit ourselves to established neural network architectures in the following. The interested reader may find additional resources on other approaches in the literature (e.g. [1; 5; 13]).

2.1.1 Recurrent Neural Network Architectures

ANNs for time series modeling are typically realized as recurrent architectures. They relax the constraint of acyclic computational graphs which is imposed on traditional feedforward networks like Multi Layer Perceptrons (MLP). In an MLP, any input of the network is directly mapped to an output. If the acyclic condition is relaxed, each output may be influenced by activations of previous inputs, resulting in the theoretic possibility of considering the entire history of previous inputs for some output. This way, recurrent networks obtain a memory, allowing this kind of networks to process sequences of values [13, p. 20]. Figure 2.2 shows a part of a recurrent neural network with a cyclic connection that feeds the output back into the layer together with the input at a given time step.

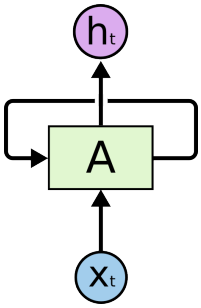


Figure 2.1: Recurrent Neural Network with one hidden layer. Source: [19]

To visualize what happens when allowing loops in the computational graph, the cyclic graph can be unfolded into an acyclic graph [13, p. 20 f.]. Figure 2.2 shows the acyclic unfolded graph that corresponds to the cyclic graph shown in figure 2.1. The activation of the hidden layer is now no longer a result of only the external input but also of the activation from the same layer at previous time steps [13, p. 19 f.].

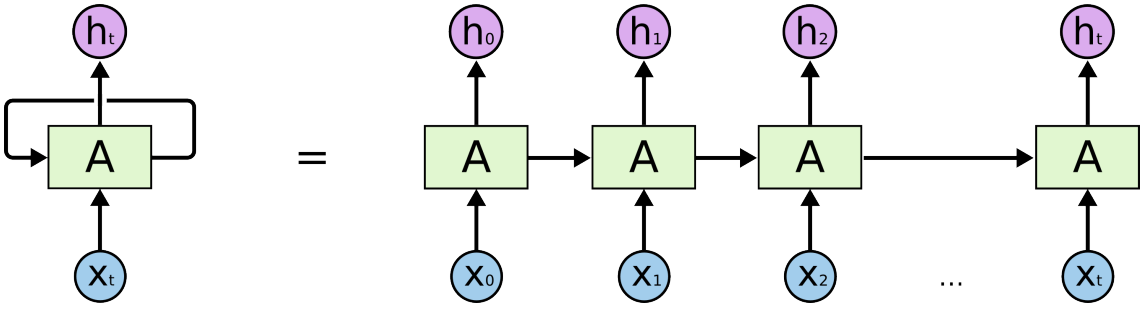


Figure 2.2: Unfolded Recurrent Neural Network. Source: [19]

Despite the theoretical possibility to consider an infinite history, RNNs exhibit in practice a limited context range. This is a consequence of the cyclic architecture which may lead to either exploding or vanishing influence of some input on the network output [13, p. 31]. As a solution, gated RNNs have been introduced. Instead of directly looping back in the network graph to accumulate knowledge over time, gates are used to steer the flow of information over time. These gates loop back with weights that depend on the context

rather than using fixed weights [12, p. 397 ff.].
One such gated network architecture called *Long Short-Term Memory* (LSTM) was introduced in 1997 by Hochreiter and Schmidhuber [14]. The LSTM architecture is depicted in figure 2.3. The upper horizontal line is the cell state which serves as memory cell of the LSTM. It can be reset, written and read through three different gates: the *forget gate*, the *input gate* and the *output gate* [13, p. 33]. Gates are activated by (1) the output and state of the LSTM block corresponding to the previous time step and (2) the external input of the current time step. The self-loop weight of the cell state is controlled by the forget gate, which is depicted as the first hidden *sigmoid* layer of figure 2.3. It outputs weights between 0 and 1 which are then multiplied with the cell state and thus steers which information in the cell is preserved or forgotten. The input gate then decides which information to add to the cell state, shown in figure 2.3 as the second *sigmoid* layer. A *tanh* layer creates new candidates to add to the cell state. Finally, the output gate can block the output of the LSTM, shown in the figure as the *sigmoid* layer on the far right.

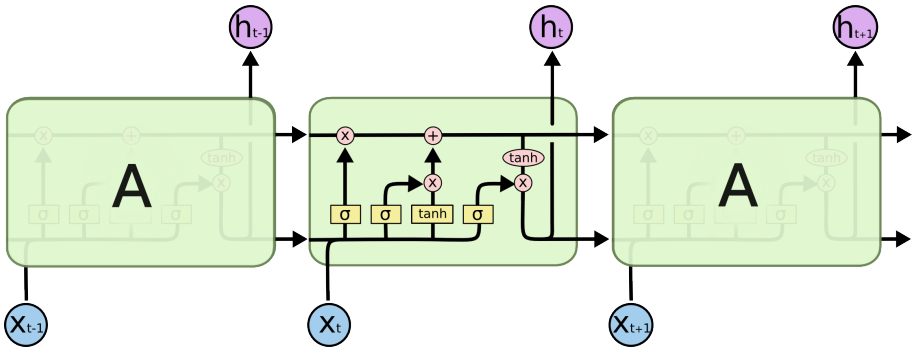


Figure 2.3: Architecture of the Long Short-Term Memory. Source: [19]

Another more recent gated architecture is the *Gated Recurrent Unit* (GRU) introduced in 2014 by Cho et al. [7]. One main difference to LSTMs is that input and forget gates are merged into a single *update gate* and that there is no separate output from the cell state [12, p. 400 f.].

2.1.2 Temporal Convolutional Networks

RNN architectures as explained in the previous section are often regarded as the natural approach for time series modeling with ANNs due to their memory capacity. By feeding the output back to the input, parameters are shared across different time steps of the sequence. Another approach that shares weights across time steps are time-delay neural networks (TDNN) [12, p. 364]. TDNNs were introduced in 1989 by Waibel et al. [23] and perform in principle a convolution[1] across the input and activations of intermediate hidden layers for successive time delays (hence the name). However, while the unfolded recurring architecture of RNNs as discussed before can result in very deep computational graphs,

[1] for an explanation of the convolution operation, see paragraph *Convolution* later in this section

TDNNs remain relatively shallow due to the small neighborhood that the convolutional kernels consider [12, p. 364].
Recently, architectures using one-dimensional convolutions have been proposed again for time-series modeling. One prominent example is the waveform generation network WaveNet by van der Oord et al. [20]. The problem of small receptive fields that TDNNs suffer from is addressed by using *dilated convolutions*. In contrast to traditional one-dimensional convolutions, a dilated convolution applies the kernel not to adjacent inputs but skips values of the input with a certain step width. A dilation of 1 is equivalent to a regular convolution. In the WaveNet architecture, the dilation value is increased exponentially with each layer in the network, resulting in a large receptive field without a too deep architecture or too large convolutional kernels. This principle can be seen in figure 2.4, where a neuron in the top layer has a receptive field size of 16 time steps with only four convolutional layers and a kernel size of 2. Without dilation, the receptive field size would shrink to 5 time steps in this layer.

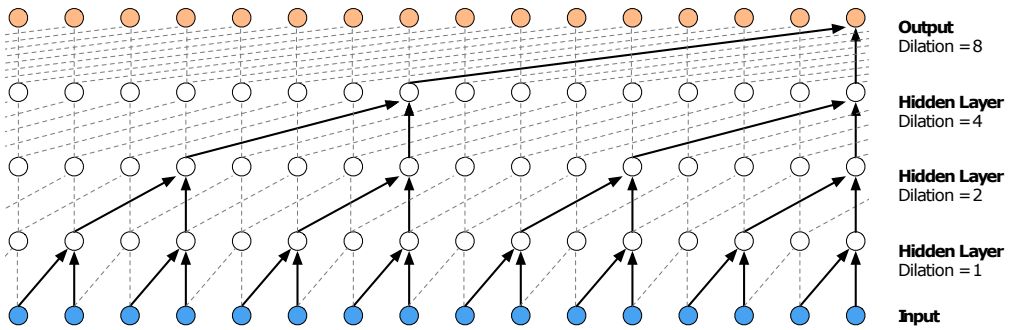


Figure 2.4: Dilated convolution with kernel size 2 and exponential dilation growth. Source: [20]

The term *Temporal Convolutional Network* (TCN) for a hierarchy of temporal dilated convolutions was coined by Lea et al. [17] and is based on the aforementioned WaveNet architecture. Bai et al. [3] describe a general TCN architecture inspired by WaveNet and evaluate its performance on several benchmarks against RNNs, GRUs and LSTMs. Their results suggest that TCNs are well suited for a broad range of tasks, exhibiting a longer memory than recurrent architectures. At the same time, the authors find the TCN architecture to be simpler and in many cases more accurate than recurrent models. As a result, they suggest that "convolutional architectures should be regarded as a natural starting point for sequence modeling" [3]. In the following section, we describe this general TCN architecture which is also the one we use throughout the experiments of our work.

Architecture

The general TCN architecture introduced by Bai et al. [3] uses at its core dilated causal convolutions and can take sequences of arbitrary length as input. Causal means that a

convolution at some time step is not performed on input values of future time steps but may only consider the past of the input sequence.
The building block that the authors propose is called a *residual block* and a TCN architecture essentially consists of a number of stacked residual blocks. The structure of these blocks can be observed in figure 2.5. A residual block n takes as input the output activation $\mathbf{out^{(n-1)}}$ of the previous block $n-1$, which is a vector of the same length T as the input sequence. The first residual block simply takes the original time series as input. Inside the residual block, the input is fed to two layers of dilated convolution which both use weight normalization and are followed by a rectified linear activation activation (ReLU) and a dropout layer.
The result of the convolutional branch is added to the original input by a parallel branch that applies no transformation in order to allow the block to learn a manipulation to the input instead of a pure transformation. Such a parallel branch that applies no transformation is called a *skip connection* [20]. The optional (1×1) convolution in this skip connection is necessary in cases where the dilated convolutional layers increase the number of channels. In this case, to allow summation of both branches in the block, the input must be inflated to the same dimensionality as the activation of the convolutional layers in the block, which is achieved through a (1×1) convolution. We highlight that in addition to the described architecture of Bai et al., in their implementation[2] the authors placed an additional ReLU nonlinearity behind each residual block that is not explicitely mentioned in the paper. Throughout all evaluations in our own research, we keep this additional ReLU layer.
Note that both dilated causal convolutional layers within one residual block have the same kernel size and dilation factor. While the kernel size is kept the same for all blocks in the network, the dilation is doubled with each successive residual block. In order to keep the length of the block activation the same as the input sequence, zero padding of length $(k-1)$ is applied for each convolutional layer, where k is the kernel size of the layer. As a consequence, the output of a stack of residual blocks has always the same length as the input sequence.

Convolution

At the core of a TCN lies the convolutional operation that each neuron in a layer applies to all channels that are fed into the layer. The output of a neuron is generated by convolving the learned convolutional kernel with the input channels.
The convolution of two functions u and v is defined as the infinite integral over the product of these two functions after reversing and shifting one of them. Mathematically, this commutative operation is expressed for the one-dimensional case as follows [2, p. 176]:

$$(u * v)(x) = \int_R u(x-y) \cdot v(y) dy = \int_R u(y) \cdot v(x-y) dy. \tag{2.1}$$

[2] the TCN implementation corresponding to the paper by Bai et al. can be found at `https://github.com/locuslab/TCN`

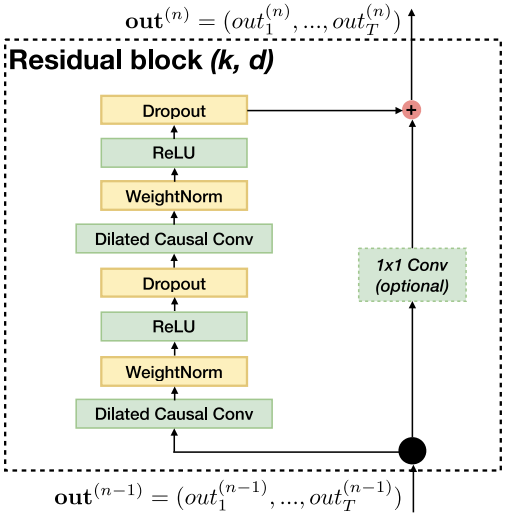


Figure 2.5: Residual Block in the TCN architecture. Adapted from Bai et al. [3]

Similarly, the convolution of two discrete signals can be formulated as follows:

$$(u * v)[n] = \sum_{m=-\infty}^{\infty} u[n-m] \cdot v[m] = \sum_{m=-\infty}^{\infty} u[m] \cdot v[n-m] \tag{2.2}$$

Accordingly, a dilated convolution with dilation d (denoted as $*_d$) is defined through [3]:

$$(u *_d v)[n] = \sum_{m=0}^{k-1} v[m] \cdot u[n - d \cdot m] \tag{2.3}$$

In practice, the authors of [3] use the class *Conv1d* from the PyTorch library. Instead of using the convolution operation, this class uses the cross-correlation function[3], which is defined as [2]:

$$(u \star v)(x) = \int_{R^n} u(y) \cdot v(y + x) dy. \tag{2.4}$$

As one can see from comparing (2.1) and (2.4), cross-correlation differs from convolution in that neither function is reversed. If we see u as the input series and v as the kernel of the convolutional kernel, cross-correlation and convolution may be used equivalently in the scope of TCNs. This is because the kernel values of v are learned from the training data and using cross-correlation instead of convolution simply results in a reversed order of the learned kernel weights.

Receptive Field Size Calculation

As mentioned, an important characteristic for any time series model is the capability to consider the past for predicting values of the future. As described above, Temporal Con-

[3] see `https://pytorch.org/docs/stable/nn.html#conv1d`

volutional Networks use the concept of dilated convolutions to achieve a larger receptive field. By dilating the kernel exponentially in each residual block, the receptive field and thus the extent to which the model considers time values from the past, increases exponentially as well. To configure the setup of a TCN for a given task and dataset, it is important to know the receptive field of the architecture. The receptive field of a neuron in a given layer is the number of time steps from the original input series that the neuron is path-connected to [18]. In the following, we derive the formulas to calculate the receptive field for an arbitrary layer or residual block in a Temporal Convolutional Network.[4]
The receptive field size $F(l)$ of a given layer l depends on the kernel size k_l and the dilation d_l of this layer as well as the receptive field $F(l-1)$ of the layer below. From figure 2.4, one can easily see that the receptive field of layer l spans across $((k_l-1)\cdot d_l+1)$ time steps of layer $(l-1)$, i.e. the layer *before*. Due to the overlap of one time step, an arbitrary layer l therefore increases the receptive field of the layer before by size $F_l'(l)$:

$$F_l'(l) = (k_l - 1) \cdot d_l \tag{2.5}$$

Given the enhancement of the receptive field size with layer l, we can express the total receptive field size at a layer l through the recursive function

$$F_l(l) = F_l'(l) + F_l(l-1) \tag{2.6}$$

and we define

$$F_l(0) := 1 \tag{2.7}$$

for layer 0 as the input series which can be seen as the result of a (1×1) convolution. By expanding (2.6), we obtain

$$\begin{aligned} F_l(l) &= F_l'(l) + F_l(l-1) \\ &= F_l'(l) + F_l'(l-1) + F_l(l-2) \\ &= F_l'(l) + F_l'(l-1) + F_l'(l-2) + ... + F_l(0) \\ &= F_l(0) + \sum_{j=1}^{l} F_l'(j) \end{aligned} \tag{2.8}$$

and using (2.5), this results in

$$F_l(l) = F_l(0) + \sum_{j=1}^{l} (k_j - 1) \cdot d_j \tag{2.9}$$

By design of the architecture, kernel sizes of all layers are the same, i.e. $k_l = k = const.$

[4] A good explanation for the calculation of the receptive field size can also be found at `https://medium.com/the-artificial-impostor/notes-understanding-tensorflow-part-3-7f6633fcc7c7`

and we can further simplify the expression. Using (2.7), this yields

$$F_l(l) = 1 + (k-1)\sum_{j=1}^{l} d_j \tag{2.10}$$

which is an expression for the receptive field size of any layer l in the network. From this equation, we now derive the receptive field size as a function of the residual block n in the network. The particularity of a residual block is that it contains two stacked convolutional layers that have the same dilation. To account for this, we can express the dilation of any layer in the network as follows:

$$d_l = \begin{cases} 2^{\frac{l-1}{2}} & \text{for } l = 1, 3, 5, 7, 9... \\ d_{l-1} & \text{for } l = 2, 4, 6, 8, 10, ... \end{cases} \tag{2.11}$$

Accounting for the fact that each residual block contains two convolutional layers, the receptive field size $F_n(n)$ for block n can be obtained from (2.10) using (2.11):

$$\begin{aligned} F_n(n) &= F_l(2n) \\ &= 1 + (k-1)\sum_{j=1}^{2n} d_j \\ &= 1 + (k-1)\sum_{j=1}^{n} d_{2j-1} + d_{2j} \\ &= 1 + (k-1)\sum_{j=1}^{n} 2 \cdot d_{2j-1} \\ &= 1 + (k-1)\sum_{j=1}^{n} 2 \cdot 2^{\frac{2j-2}{2}} \\ &= 1 + (k-1)\sum_{j=1}^{n} 2^j \\ &= 1 + (k-1) \cdot (2^{n+1} - 1) \end{aligned} \tag{2.12}$$

Using equations (2.10) and (2.12), we can now calculate the size of the receptive field for any layer or block in the network respectively.

2.2 Transfer Learning

Convolutional neural networks have achieved remarkable results on a broad variety of tasks and datasets in the past, especially in the area of Machine Vision. Typically, these results are achieved by leveraging vast amounts of data that needs to be collected beforehand and that can be used as training data. In case of supervised learning or semi-supervised learning, this data needs to be (partly) labeled before the training process and the amount

of available training data must be sufficient to train a model. An example for a large and well-studied dataset is the ImageNet database [9]. However, the availability of (labeled) data is typically not a given. Another assumption for most training approaches is that the training data and the test or application data are drawn from the same distribution [21]. The traditional machine learning approach therefore requires the generation of a new model for every new task or domain.
The approach of Transfer Learning aims to relax either or both of these conditions. The goal is to obtain a model even if one of the assumptions of traditional machine learning does not hold anymore, i.e. if for some task on some domain there is not enough data available to train a model from scratch or if training and test data cannot be assumed to be drawn from the same distribution. The idea is to extract knowledge from a model that was trained on some source task using data from a source domain and to apply this knowledge to a target task on a target domain. This way, Transfer Learning allows to generate models where otherwise collecting training data would be too expensive or impossible. Another important benefit of Transfer Learning is the faster generation of models if the model does not need to be trained entirely from scratch [21].

We use the notions of the terms *domain* and *task* that Pan et al. introduce in their survey on Transfer Learning [21]. In this survey, the authors define a domain $\mathcal{D} = \{\mathcal{X}, P(X)\}$ as a tuple of a feature space $\mathcal{X}$ and a marginal probability $P(X)$ over this feature space. $X = \{x_1, ..., x_n\} \in \mathcal{X}$ is a set of samples where x_i is the feature vector of the i-th sample. A corresponding task $\mathcal{T} = \{\mathcal{Y}, f(\cdot)\}$ is respectively defined as a tuple of a label space $\mathcal{Y}$ and a prediction function $f(\cdot)$. The prediction function $f(\cdot)$ can be understood as the conditional probability $P(Y|X)$ that $Y \in \mathcal{Y}$ is the set of true labels for samples $X \in \mathcal{X}$. It is learned from the training data that contains pairs of $\{x_i, y_i\}$, where $x_i \in \mathcal{X}$ and $y_i \in \mathcal{Y}$. Pan et al. define the goal of Transfer Learning to help improve the learning of some target prediction function $f_T(\cdot)$ of target task $\mathcal{T}_T$ on target domain $\mathcal{D}_T$ using the knowledge from a source task $\mathcal{T}_S$ on a source domain $\mathcal{D}_S$, where $\mathcal{D}_S \neq \mathcal{D}_T$ or $\mathcal{T}_S \neq \mathcal{T}_T$. A transfer between different tasks is referred to as *inductive transfer* while a transfer between identical tasks is called *transductive*. The following four scenarios are possible:

1. The source and target domains live in different feature spaces, i.e. $\mathcal{X}_S \neq \mathcal{X}_T$
2. The source and target domains have different marginal probability distributions, i.e. $P(X_S) \neq P(X_T)$
3. The source and target tasks have different label spaces, i.e. $\mathcal{Y}_S \neq \mathcal{Y}_T$
4. The source and target tasks have different conditional probability distributions, i.e. $P(Y_S|X_S) \neq P(Y_T|X_T)$

For a succesful transfer, source and target domain must be related. In addition, models contain knowledge specific to domains and tasks and knowledge that can be regarded as general. The latter may be beneficial for obtaining a model on the target domain or task

[21; 7, p. 166; 24]. If domains are not related or the transferred knowledge is too specific to the source task, this may hurt the performance of the target model, resulting in a *negative transfer* [21].

Chollet [7] describes two approaches to transfer knowledge from a pretrained network. The first approach, called *feature extraction*, uses representations from a model trained on a source domain and task to extract features from data in the target domain that are interesting to solve the target task. The general approach for feature extraction is to transfer and freeze weights from the source model and to use them in the target model [7, p. 143 ff.]. The second approach is called *fine-tuning* and also consists in transferring weights from the source model to the target model. However, unlike in the case of feature extraction, weights are not frozen but may train together with the rest of the target model on the target domain and task [7, p. 152 ff.].

Both Pan et al. [21] and Chollet [7] highlight that knowledge in a model can be general or specific and that domains must be related for a successful transfer. It is therefore desirable to know which parts of a network can be regarded as general or specific and are thus well transferable. Yosinski et al. [24] evaluate the degree of transferability for different layers in CNN architectures. Starting from the observation that low layers learn general features whereas deep layers learn specific features, they posit that there must be a transition from general to specific somewhere in the architecture of the network. The authors aim at quantifying the degree of transferability of particular layers by assessing how general or specific they are with regards to the task and domain of the model. At the same time, they investigate into where and how sudden the transition takes place in the network.
To address these points, the authors propose an experiment where two models are trained on two distinct datasets. The initial models are referred to as the *base models*. Both models are then chopped at some level of the network by reinitializing the top layers with random weights. In a next step, both models continue training on the same one of the initial two datasets, i.e. the target dataset. As a consequence, one of the models (called the *selffer model*) is retrained on previously seen data while the other sees a different dataset. The latter - called *transfer model* - hence performs a domain transfer from one dataset to the other. The intuition behind this experiment is that the selffer model's accuracy on the test set should not change much compared to the base model while the transfer accuracy is expected to drop at a certain level of the network. This drop can then be regarded as a transition of layers from being general to being specific.
The experiment is repeated both with a feature extraction and a fine-tuning setup. The authors use the ImageNet dataset which they split randomly into two parts with distinct labels to obtain separate datasets. The setup can therefore be characterized as an inductive transfer because $\mathcal{Y}_S \neq \mathcal{Y}_T \Rightarrow \mathcal{T}_S \neq \mathcal{T}_T$.
Yosinski et al. [24] observe in their experiments that the transfer accuracy indeed drops at a certain level in the network below the selffer and base model accuracy. However, besides

this expected behaviour, they also find that the selffer model shows decreased performance when chopped at certain adjacent layers. This hints at what the authors refer to as *fragile co-adaptation of layers*, meaning that the features of these layers interact with each other. Furthermore, the authors find that a transfer model may even experience a performance boost compared to the base model if transferred layers are not frozen but can be fine-tuned on the target dataset.

2.3 Visualization of Convolutional Neural Networks

For Transfer Learning to benefit the target task and to avoid a negative transfer, source and target datasets are required to be similar or *related*. In other words, the knowledge to be transferred that is encoded in a source model must be of interest for the target task on the target domain. As said, this kind of information can be regarded as general for both datasets and may thus benefit the generation of a model on the target domain for the target task. Approaches for evaluating the transferability generality of features in a CNN were presented in the previous section. However, even if the degree of transferability and specificity of the knowledge encoded in a network are known, this does not reveal what kind of information is encoded in the network. To learn what it is that convolutional networks learn, different visualization approaches have been proposed in the literature.
The most straightforward way to visualize the knowledge in a CNN is by plotting the activation of intermediate layers of the network. Since each neuron in a typical CNN layer produces an two-dimensional output, called *activation*, this output can be interpreted as an image that may be directly plotted. Because an activation can only be observed when the network is excited with some input, this visualization is *input-specific*. Typically, low layers in computer vision CNNs learn very general features like edge detectors. Activations deeper in the network become more abstract and specific to the task and domain. At the same time, activations deeper in the network tend to be sparse, i.e. not all neurons are activated by some input [7, p. 160 ff.].
Another visualizing technique for CNNs is to plot the convolutional filter kernels that the network has learned. For this approach, weights of two-dimensional convolutional kernels in a CNN are interpreted as pixel values. This technique reveals, like the activation visualization, the behaviour of CNNs to learn general features in low layers [16; 6]. Unlike the activation, visualized kernel weights are independent of the input. However, this technique usually produces well interpretable results only for low layers [25].
A more elaborate way of visualizing what CNNs learn was introduced by Erhan et al. as the *activation maximization* approach [11]. The two previous approaches visualize the activation and kernels *as if* they lived in the input space, e.g. as a two-dimensional image. However, in fact neither activation nor kernel weights actually exist in this space. The activation maximization on the other hand aims to find an instance from the input space that maximizes the activation of a specific hidden unit in the network. For this purpose, gradient ascent is performed on a random input with bounded norm to maximize the

activation of some hidden unit of interest. Gradients of the activation are computed with regards to the input and the model's parameters. The input is then modified to maximize the activation while model parameters are kept fixed. The activation maximization technique shows that deep units represent more complicated features which are composed of lower layer features. A caveat of the approach is that it only converges well for relatively small inputs.
Inspired by the work of Erhan et al., Zeiler and Fergus [25] introduce another approach to visualize CNN knowledge in input space, but without the convergence problem for large inputs. Their approach uses deconvolutional networks [26] to project the activation of a unit back into input space. The deconvolutional network performs a projection from an arbitrary hidden unit of interest in the network back to input space by inverting the operations of all layers in the network up to that hidden unit. The inverse of the convolutional operation is approximated by convolving with the transposed of the learned kernels.

2.4 Synthetic Data Generation

The goal of this work is to find out how transferable layers in TCNs, what knowledge they learn and how much knowledge the different layers contribute to overall model performance. For all these aspects, we design and run experiments that require training and testing of TCN models which we need adequate time series datasets for. To obtain a dataset for the experiments, either publicly available datasets of real-world time series[5] or synthetically generated time series data can be used. Real-world datasets hold the advantage that they naturally contain characteristics that any practically applied model may be exposed to, such as trends, seasonality, random artifacts and more. Another advantage is the availability of benchmarks for many of these datasets. On the other hand, said characteristics may also result in unexpected behaviour if not all of the characteristics are known. A synthetic dataset on the other hand holds the advantage that the difficulty of the dataset to be learned by the model can be adjusted. Such a series can be designed to incorporate or explicitly not have specific characteristics.
Since the field of Transfer Learning has not been researched in depth yet for TCNs, we choose to run our experiments with synthetic datasets instead of using real-world data. This allows us to make sure that the data can be learned by our model and to find a set of hyperparameters that works well. At the same time, generating our own data, we can be certain that the data is not trivial to learn, i.e. the model converges too fast. However, we highlight that parts of the results therefore may not be representative for some practical applications of TCNs and that real-world datasets should be regarded as a natural next step for further evaluation of TCNs. In the following, we explain how we generate the synthetic dataset used throughout this work.
To obtain useful training and test data, we superimpose a number of sinusoidal functions. Since any time series can be decomposed into a sum of sinusoidal functions [4], arbitrary

[5] such datasets are available for example on the UCI Machine Learning Repository. See `https://archive.ics.uci.edu/ml/datasets.php`

signals can be generated by appropriately superimposing a sum of sine functions. We therefore posit that the approach of superimposing sine signals is well suited to generate a signal with sufficient difficulty for our model to learn. The difficulty to learn the dataset can be steered by the number of sine functions that are superimposed.
For the experiments in this work, we generate two time series of length $len = 10000$ by superimposing a total of 150 sine functions with different frequencies and amplitudes, resulting in the following sum:

$$x[t] = \sum_{s=1}^{150} sin(freq_s \cdot 2\pi t) \cdot amp_s \tag{2.13}$$

Where $x[t]$ is the value of the series at time step t, $freq_s$ is the temporal frequency of the s-th superposition and amp_s the s-th amplitude. According to the Nyquist-Shannon sampling theorem, if a continuous signal, containing only frequencies smaller than some maximum frequency $freq_{max}$, is sampled at $freq_{sample} = 2freq_{max}$, all information from the continuous signal is preserved in the sampled signal [22]. We choose $t \in \mathbf{N}^+$ which means that $freq_{sample}$ is 1. The maximum frequency present in the generated signal should therefore ideally be

$$freq_{max} = 0.5 \cdot freq_{sample} = 0.5 \tag{2.14}$$

To ensure that the created time series are not always the same, both the s-th frequency and amplitude are drawn from normal distributions. We set the expectation value for $freq_s$ to

$$\mathbb{E}[freq_s] = \frac{(freq_{max} \cdot len)^{(s-1)/149}}{len} \tag{2.15}$$

and its standard deviation to $\frac{1}{3}\mathbb{E}[freq_s]$. This way, frequencies are likely to fit into the sequence with at least one complete sinusoidal cycle and to have a maximum frequency around $freq_{max}$. At the same time, both very large and very small frequencies are contained in the overall series. The amplitude is drawn from a normal distribution with mean 0 and standard deviation $\frac{1}{3}$. In addition, we add random noise to the signal that is bounded between 0 and 5% of the signal amplitude.

The series shall be used for an autoregression task, i.e. models will be trained to predict the future of the series given its past. To formulate such a prediction task for the synthetically generated dataset, target values y for each value in the source series x are generated through shifting the series by 1, such that for values in the source and the target series we obtain the following relation:

$$y[t] = x[t + 1] \tag{2.16}$$

The goal of the model is then to learn a prediction function $f(\cdot)$ that maps past values of

x to the current value of y in the following manner:

$$\hat{y}[t] = f(x[t], x[t-1], ..., x[t-T]) \approx y[t] \tag{2.17}$$

where $\hat{y}[t]$ is the predicted value of the model, T is the most distant past of x to be considered and $y[t]$ is the true value. In other words, the task consists in predicting the value at the next time step of the series given the past of this series.

According to the outlined procedure, we generate two datasets A and B which we use throughout our experiments. Both of these datasets can be seen as realizations from two domains $\mathcal{D}_A = \{\mathcal{X}_A, P(A)\}$ and $\mathcal{D}_B = \{\mathcal{X}_B, P(B)\}$ where $\mathcal{X}_A$ and $\mathcal{X}_B$ are the feature spaces in which the datasets live and $P(A)$ and $P(B)$ their respective marginal probability distributions over these feature spaces.

Chapter 3

Qualitative Evaluation of Layer Transferability

Based on the reviewed literature and theoretical foundations from the previous chapter, this chapter focuses on the first research question. The experiments conducted to answer the qualitative question of whether layer knowledge in TCN models is transferable are to a large extent inspired by the transferability experiment of Yosinski et al. [24]. To assess the transferability for TCNs, we run three main experiments and a reference experiment, which are explained in the first section 3.1 of this chapter. The training and evaluation process to run and interpret these experiments is explained in section 3.2. Results and interpretations for all experiments as well as an evaluation of training durations are presented in section 3.3. We conclude the chapter with a comparison and joint discussion of all obtained results in section 3.4.

3.1 Experiment Overview

To evaluate the transferability of layers in Temporal Convolutional Networks, we run three main experiments and one reference experiment. The idea of the three main experiments is derived from the work of Yosinski et al. [24]. We train two *base models* $\mathcal{M}_T$ and $\mathcal{M}_S$ with identical architecture, one on the target domain $\mathcal{D}_T$ and one on the source domain $\mathcal{D}_S$. As datasets, we use the two generated series A and B as explained in section 2.4 of the previous chapter. We then transfer layers of the pre-trained network $\mathcal{M}_S$ to a new, untrained network $\mathcal{M}_{S\to T}$ and train this new network on the dataset from domain $\mathcal{D}_T$. This *transfer model* $\mathcal{M}_{S\to T}$ performs a transfer from the source domain $\mathcal{D}_S$ to the target domain $\mathcal{D}_T$. Equivalently, layers from $\mathcal{M}_T$ are transferred to a new, untrained model $\mathcal{M}_{T\to T}$ which is then trained on $\mathcal{D}_T$ as well. This latter model, called *selffer model*, serves as a comparison for the transfer model. A performance difference between transfer, selffer and base models can then be used as an indicator for the success of the transfer. We run the general setup in three flavours that form the three main experiments as explained in the follwing section. We also run the experiments of all three flavors in both directions of datasets, i.e. with A as source dataset and B as target dataset and vice versa. In the following, the default direction is the latter case with B as source dataset and A as target

dataset. Whenever the reverse transfer direction yields results worth noting, these will be mentioned and shown explicitly. All other results can be found in appendix A.1.

3.1.1 Main Experiments

Yosinski et al. [24] run two variations of their experiment. One variation keeps the transferred layers frozen for training on $\mathcal{D}_T$, while the other one also allows transferred layers to train on the target domain. In both variations, upper layers that were not transferred are always initialized with random weights. Their experiments hence differ in whether transferred layers are frozen or not. We add another variation parameter to the experiment by initializing upper layers with either random or pre-trained weights from the base model. This results in the possible experiments shown in table 3.1. The experiments are repeated with different values for the parameter $l \in [1, L]$, where L is the total number of convolutional layers in the network architecture (not to be confused with the number of residual blocks). We call l the *transfer level*. For experiments where transferred weights are frozen, the lower l weights of $\mathcal{M}_{T\to T}$ and $\mathcal{M}_{S\to T}$ are not updated when training on $\mathcal{D}_T$. For experiments that are executed with randomly initialized weights, we set the weights of all upper $(L - l)$ layers to random values. We refer to these experiments as *feature extraction*, which is either *hard* in the case of frozen layers or *soft* when layers are left unfrozen. We call experiments where upper layers are not initialized randomly a *complete weight initialization*, since the whole network is transferred. The initialization can either happen with or without freezing the lower l layers. Note the following points that result from this configuration space:

- Experiments 1 and 2 have identical corner cases for $l = 0$. No weights are transferred, leaving all layers with random initialization.
- Experiments 1 and 3 have identical corner cases for $l = L$. All weights are transferred and completely frozen.
- Experiment 3 for $l = 0$ and experiment 2 for $l = L$ are identical corner cases. Weights are completely transferred without freeze.
- Experiment 4 does not change with l, since independently of l, all weights are transferred and none are frozen. In particular, this case represents a corner case of experiment 3 for $l = 0$ or experiment 2 for $l = L$. Therefore, this experiment is not separately considered.

Excluding the fourth experiment yields the three main experiments *Hard Feature Extraction*, *Soft Feature Extraction* and *Full Weight Initialization with partial Freeze*. All three experiments represent a different type of knowledge transfer from domain $\mathcal{D}_S$ to domain $\mathcal{D}_T$.
We first train two networks $\mathcal{M}_T$ and $\mathcal{M}_S$ on $\mathcal{D}_T$ and $\mathcal{D}_S$ respectively. These networks are referred to as *base models*. From both base models, we successively transfer layers at different transfer levels to two new models $\mathcal{M}_{T\to T}$ and $\mathcal{M}_{S\to T}$. Both are then trained

Table 3.1: Experiment table for the qualitative transferability evaluation. Names in quotes indicate respective experiments in the evaluation of Yosinski et al. [24].

	Freeze (layers $\leq l$)	**Rand init** (layers $> l$)	**Description**
1	True	True	Hard Feature Extraction (`"BnA"`)
2	False	True	Soft Feature Extraction (`"BnA+"`)
3	True	False	Full Weight Initialization with partial Freeze
4	False	False	Full Weight Initialization without Freeze

on the target domain $\mathcal{D}_T$. We use the terminology from Yosinski et al. [24] and refer to $\mathcal{M}_{T \to T}$ as the *selffer* network because the layers are retrained on the sama data as before. This network serves as a comparison to make sure that a performance improvement is not simply due to a longer training time but is a result of the knowledge transfer. $\mathcal{M}_{S \to T}$ on the other hand is the *transfer* network which we compare against the initial base network. The expectation is that the selffer network will show a similar performance as the base network because it is not trained on any new training data. If the selffer model performance decreases for some transfer levels, this could be considered an indication for fragile co-adaptation. For the transfer network on the other hand, informed by the results from Yosinski et al., we expect a performance drop at some transfer level which marks the transition from general features to specific features. Figure 3.1 shows a visualization of the two base networks (rows one and two) as well as the transfer networks of the different main experiments (rows three to five) and the reference experiment (row six).

The first experiment is the equivalent to the `BnA` experiment conducted by Yosinski et al. [24]. Layers up to the transfer level l are transferred to the target network and frozen, while all other layers are initialized with random weights. An example for a resulting model with transfer level 5 is shown in figure 3.1, row 3, where colored rectangles represent transferred weight matrices. The transferred layers can be understood as a pre-trained set of hard extracted features that originate from the base model. The remaining, unfrozen layers use the output features of the transferred layers to abstract to features specific to the target domain $\mathcal{D}_T$ and finally to a prediction value.

The second experiment simulates a partial weight initialization using pre-trained weights from the base model and is equivalent to the `BnA+` experiment of Yosinski et al. [24]. In this case, transferred layers in the target model are not frozen. Instead, they are fine-tuned on the target task and domain. All other layers are initialized with random values. By allowing all layers - also the transferred ones - to be updated, we shift from a Hard Feature Extraction to a soft one. The resulting model is shown in figure 3.1, row four. The idea is to fine-tune the knowledge that is contained in the transferred layers so that the weights can adapt to the target task. Even if transferred weights are potentially not directly applicable to the target task, they are expected to provide a better start for training on the target task and dataset than a random initialization.

The third experiment is an addition to the ones conducted by Yosinski et al. [24] and can be seen as a combination of the other two setups. Instead of only transferring a certain

number of layer weights to the target models, we always transfer all weights from the pre-trained model. We then freeze the lower l layers while keeping the upper layers unfrozen as shown in figure 3.1, row five. This way, the lower layers behave like a hard feature extractor in the first experiment. The upper layers on the other hand are initialized with pre-trained weights that may be assumed to be better than a random initialization.

3.1.2 Reference Experiment

Besides the previously explained three main experiments, we run an additional reference experiment that functions as a plausibility check for the main experiments. This reference experiment is identical to running experiments 1 or 3 without pre-training at all. This means that the base models $\mathcal{M}_T$ and $\mathcal{M}_S$ are set to untrained networks with random layer weights. We therefore refer to this reference experiment as the *Random Reference Model*. The main experiments aim to find out whether the knowledge learned in convolutional layers is transferable. However, whenever we allow the upper layers in $\mathcal{M}_{S\rightarrow T}$ to train on $\mathcal{D}_T$, we cannot be sure if a resulting good performance is due to the transferred and frozen weights or simply because the unfrozen part of the network leaves enough degrees of freedom to generalize on the target domain within the available training time. The random reference model aims to rule out this possibility and is similar to the evaluation with random weights by Yosinski et al. [24]. Instead of using a pre-trained model for the weight transfer, we set $\mathcal{M}_S$ to a model with complete random weights. Like in experiments 1 and 3, the lower l layers are frozen. If this setup shows a performance as good as the main experiments, we know that a good performance in the main experiments is simply due to the unfrozen part leaving enough degrees of freedom to generalize on $\mathcal{D}_T$ or to even compensate for potentially disadvantageous transferred frozen weights. If, on the other hand, the reference performs significantly less good than the transfer models, we know that the transfer performance is due to the knowledge transfer.

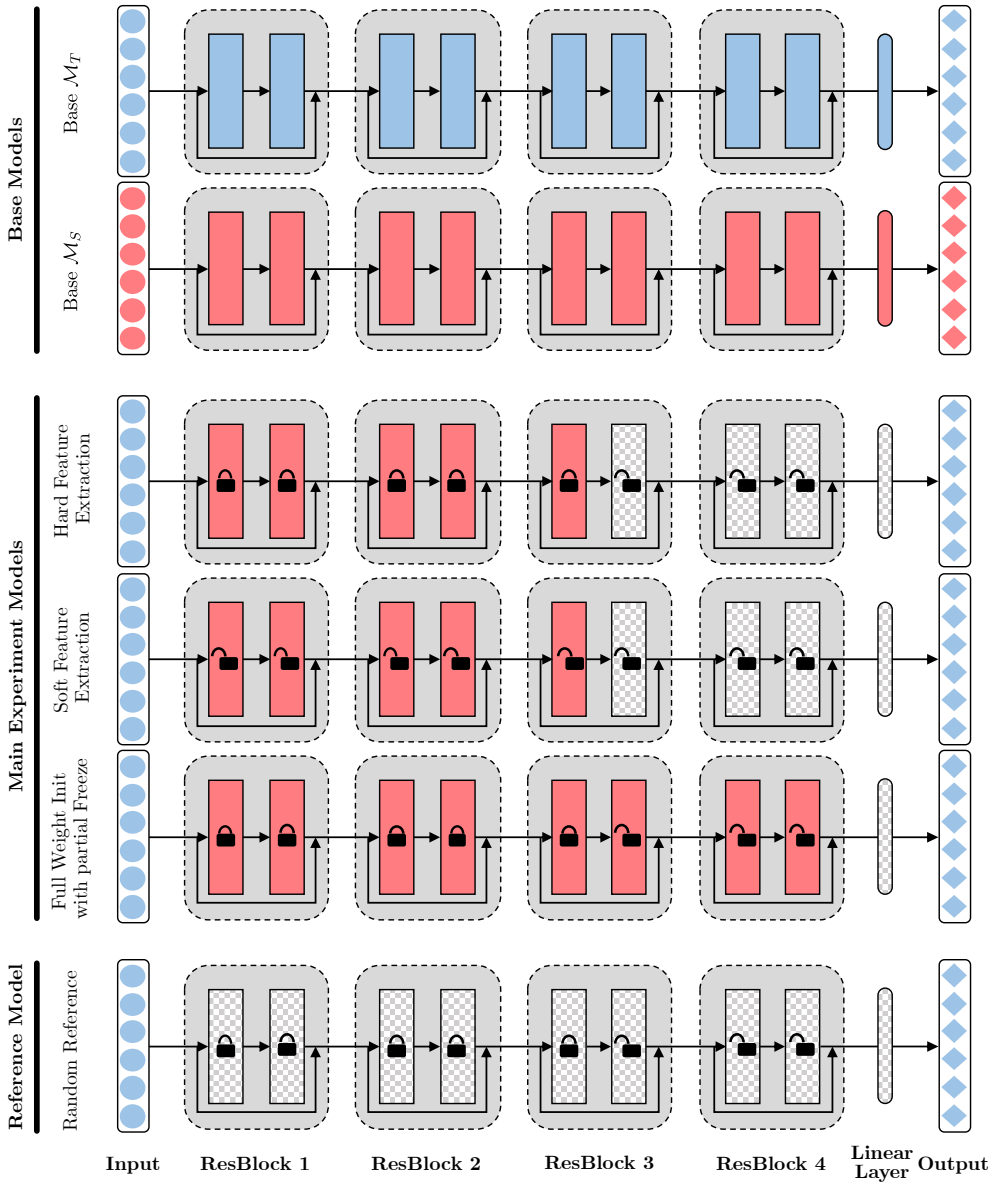


Figure 3.1: Visualization of the qualitative evaluation setup for transfer level $l = 5$. Dotted rectangles represent residual blocks, containing two convolutional layers each and a skip connection. The residual blocks are followed by a dense layer that generates the network output. The dense layer is initialized with random weights for all main experiments and the reference experiment. Selffer models are not displayed. *Top two rows*: The base networks $\mathcal{M}_T$ and $\mathcal{M}_S$ are trained on the two synthetically generated distinct datasets from $\mathcal{D}_T$ and $\mathcal{D}_S$. All layers are initialized with random weights and then trained on their respective domain. *Third row:* The lower l layers from model $\mathcal{M}_S$ are transferred and frozen. All other layers are initialized randomly. The network is then trained on dataset $\mathcal{D}_T$. *Fourth row:* Same setting as in the previous row but no layers are frozen. *Fifth row:* All layers from $\mathcal{M}_S$ are transferred but only the upper layers may learn while the lower l layers are frozen. *Sixth row:* No weights are transferred from $\mathcal{M}_S$, all layers are initialized with random weights while freezing the lower l layers.

3.2 Training and Evaluation

For all three main experiments and the reference experiment, identical training parameters and training steps are used. These parameters and training steps are described in the first two paragraphs of this section, followed by a description of how the results are evaluated.

3.2.1 Training Parameters

The data we use for this experiment are two datasets A and B generated according to section 2.4. Before the training process, the data series are normalized to have mean 0 and standard deviation 1. Empirically, we found TCNs to perform well with the set of parameters described in this paragraph. We use a dropout probability of 0.05, gradient clipping at 0.15, kernel size 6 and a total of 4 residual blocks (hence 8 convolutional layers) with 50 channels each. According to (2.12), this yields a receptive field size of 156 time steps. We perform a 80/20 split of the generated series to obtain a training and test dataset. The model is then trained for 120 epochs with an an initial learning rate of 0.0001. The learning rate is divided by 10 after epochs 100 and 110. Training batches are created with a size of 12 sequences per batch and a sequence length of 150 samples each, thus making sure that the receptive field of neurons in the deepest layer spans across the entire input sequence.

3.2.2 Training Steps

In the previous paragraph, we gave an overview on the hyperparameters that we train our models with. In the following, we introduce the steps during each of the experiments. The general pipeline is identical for all three main experiments and with a slight alteration also for the random reference experiment. To reduce the potential influence of outliers, we repeat each experiment ten times and compute the average over all repetitions. We also perform each transfer in both possible directions with regards to the domains, i.e. we transfer from domain A to domain B and from domain B to domain A for each experiment. The random reference experiment differs from this in that there is no actual transfer of learned weights. Instead, random, untrained weights are 'transferred' to the domains of dataset A or B respectively. Each of the ten repetitions comprises the following steps:

1. **Training of two base models $\mathcal{M}_T$ and $\mathcal{M}_S$ on the two datasets.** Note that for reasons of consistency, we always refer to $\mathcal{M}_T$ as the model that is trained on the dataset of the *target domain* and to $\mathcal{M}_S$ as the model that is trained on the *source* domain. Therefore, $\mathcal{M}_T$ may be trained on dataset A or dataset B, depending on the direction of transfer. The same holds true for $\mathcal{M}_S$. Each model is trained for a total of 120 epochs. However, we only keep those models that perform best on the test datasets and use these as the final base models. This way, the final base models are not necessarily trained for 120 epochs if the performance of a model after an earlier epoch is better. This is similar to doing early stopping, only in retrospective, i.e. we select the model where we could have stopped after training for a fixed number

of 120 epochs. In the special case of the random reference experiment, $\mathcal{M}_T$ and $\mathcal{M}_S$ are not trained but simply obtained from initializing two models with random weights.

2. **Generation of a selffer model $\mathcal{M}_{T\to T}$ and a transfer model $\mathcal{M}_{S\to T}$.** These models are derived from the base models according to the transfer scheme of the respective experiment. Both models $\mathcal{M}_{T\to T}$ and $\mathcal{M}_{S\to T}$ are then trained on the dataset of the target domain with the same training parameters as the base models. As before, we select in retrospective the selffer and transfer models that performed best during all 120 epochs. This step is repeated for each transfer level, i.e. for each of the eight layers in the network.

By repeating each experiment ten times and generating transfer and selffer models for every transfer level (see 2.), we generate a total number of 170 models per experiment. Training durations for all experiments on an Nvidia RTX 2080 Ti with 11GB RAM are displayed in table 3.2.

Table 3.2: Training durations of the three main experiments and the reference experiment. Values in the table are measured on a transfer from dataset B to dataset A, including ten repetitions per experiment.

Experiment	**total Duration**	**Duration / Epoch**
Hard Feature Extraction	20h 34min	60.5ms
Soft Feature Extraction	26h 37min	78.3ms
Full Weight Init, partial Freeze	20h 22min	59.9ms
Random Reference	17h 39min	51.9ms

3.2.3 Evaluation and Significance of Performance Differences

For the evaluation of all of the outlined experiments, we compare the performance of the transfer and selffer models to the original base models. Results of the three main experiments are shown in figures 3.2 through 3.9 in section 3.3 as losses of the transfer, selffer and base models on the target domain. Note that scales of the axes are adjusted to facilitate readability and therefore are not always identical. The horizontal axes show the transfer levels of selffer and transfer models for $l > 0$ and indicate base model losses for $l = 0$. For every experiment, we show the losses of all 170 models, i.e. 10 base models and 80 selffer and transfer models. Upper parts of the figures contain scatter plots with all 170 loss values. Base model losses are marked in black, selffer model losses in blue and transfer model losses in red. In the scatter plots, sellfer and transfer model losses are slightly shifted to the left and right on the horizontal axis to increase readability of the plots. Bottom parts of the figures show plots of the same information, averaged over all ten repetitions. We use this mean as metric to compare selffer and transfer model performances to the base model performance. However, since the model performance is measured in terms of MSE loss, we only have an unbounded, non-normalized metric to tell if one model is better than

some other model. We therefore need to check whether observed performance differences are statistically significant or if an observed difference is actually insignificantly small. Bottom plots in the figures therefore also include a confidence interval around the selffer and transfer model lines that indicate whether the respective mean deviates significantly from the base model mean. If mean μ_1 over 10 MSE values of some transfer or selffer model has a significant difference to mean μ_2 of the 10 base model MSE values, the confidence interval around the selffer or transfer model does not interfere with the dashed line of the base model. In case the dashed line lies within the confidence interval of either the transfer or selffer model, the respective averaged MSE of this model does not not deviate significantly from the base model average MSE.
To calculate the confidence intervals shown in the bottom figures, we use a two-sample t-test as described by Devore and Berk in [10, p. 499-509]. The test tells whether the difference of the means over two relatively small samples from distributions with unknown standard deviations is of some size Δ_0 or not. The null Hypothesis H_0 states that the mean values μ_1 and μ_2 have difference Δ_0:

$$H_0 : \mu_1 - \mu_2 = \Delta_0 \tag{3.1}$$

For the two-tailed test, the alternative hypothesis H_a states that the difference of both mean values is unequal to Δ_0:

$$H_a : \mu_1 - \mu_2 \neq \Delta_0 \tag{3.2}$$

The test statistic t for testing H_0 is:

$$t = \frac{\bar{x}_1 - \bar{x}_2 - \Delta_0}{\sqrt{\frac{s_1^2}{m_1} + \frac{s_2^2}{m_2}}} \tag{3.3}$$

where $\bar{x}_1$ and $\bar{x}_2$ are the means, s_1 and s_2 the standard deviations and m_1 and m_2 the sizes of the two samples. H_0 will be rejected with a type I error probability α for

$$t \geq t_{\frac{\alpha}{2},\nu} \quad \textbf{or} \quad t \leq -t_{\frac{\alpha}{2},\nu} \tag{3.4}$$

The margins of the rejection regions $\pm t_{\frac{\alpha}{2},\nu}$ are tabulated for known α and degrees of freedom ν of the t distribution. One such table can be found in [10, p. 797]. The number of degrees of freedom ν is computed as follows:

$$\nu = \lfloor \frac{(\frac{s_1^2}{m_1} + \frac{s_2^2}{m_2})^2}{\frac{(s_1^2/m_1)^2}{m_1 - 1} + \frac{(s_2^2/m_2)^2}{m_2 - 1}} \rfloor \tag{3.5}$$

with $\lfloor \cdot \rfloor$ the floor function. The resulting confidence interval for $\mu_1 - \mu_2$ is:

$$\bar{x}_1 - \bar{x}_2 \pm t_{\frac{\alpha}{2},\nu} \sqrt{\frac{s_1^2}{m_1} + \frac{s_2^2}{m_2}} \tag{3.6}$$

For our experiments, we want to know if the two mean values μ_1 of some selffer or transfer MSE and μ_2 of some base MSE deviate significantly. We therefore set $\Delta_0 = 0$. The type I error probability is set to $\alpha = 0.05$. Instead of using the tabulated probability values in [10], we use those in the Python Scipy package `scipy.stats.t`[1].
The test can only be applied if the samples are both drawn from a normal distribution and if the respective random variables are independent from each other, which can be judged from probability plots of the samples [10, p. 499]. If this plot shows a linear pattern, we may assume normality. However, the authors also highlight that samples of size less than 30 can show a nonlinear pattern even if they are drawn from a normal distribution and suggest to only consider very strong deviation from linearity in the probability plot as evidence against normality [10, p 216]. Probability plots for the average MSE losses can be found in appendix A.1.

3.3 Experiment Results

In this section, the results from all three main experiments and the reference experiment are presented. After discussing them separately in this section, an overall discussion follows in section 3.4.

3.3.1 Hard Feature Extraction

The results of the Hard Feature Extraction experiment are shown in figure 3.2. To evaluate the results, we compare the performance of the transfer and selffer models to the base model. From the loss plot we conclude the following:

- The blue line for the selffer model does not deviate significantly from the base model loss for any of the transfer levels. From this follows that any performance deviation of the transfer model is not simply due to the increased training time compared to the base model but results from the layer transfer.
- Differing from the results presented by Yosinski et al. [24], we do not see a significant selffer performance decrease for layers in the middle of the network. This indicates that unlike CNNs, the TCN does not exhibit fragile co-adaptation. Features in convolutional layers seem to be mostly independent and not interrelated. One potential explanation for the lack of fragile co-adaptation are the skip connections within each residual block.

[1]The documentation for the Student's t continuous random variable can be found at `https://docs.scipy.org/doc/scipy/reference/generated/scipy.stats.t.html`

- The loss of the transfer model increases constantly with the transfer level, with the biggest increase for layers 7 and 8 (the last residual block). At the same time, the loss for transfer levels 1 and 2 (the first residual block) does not deviate significantly from the base model loss. This indicates that layer features transition from general to specific in the middle of the network, i.e. residual blocks two and three in the present case. At the same time, up until layer 6, layers do not seem to be extremely specific yet, given the relatively small performance difference to the base model compared with the last residual block. This behaviour does not change qualitatively when switching the datasets which the interested reader may check in the respective figure in the appendix (cf. figure A.3).

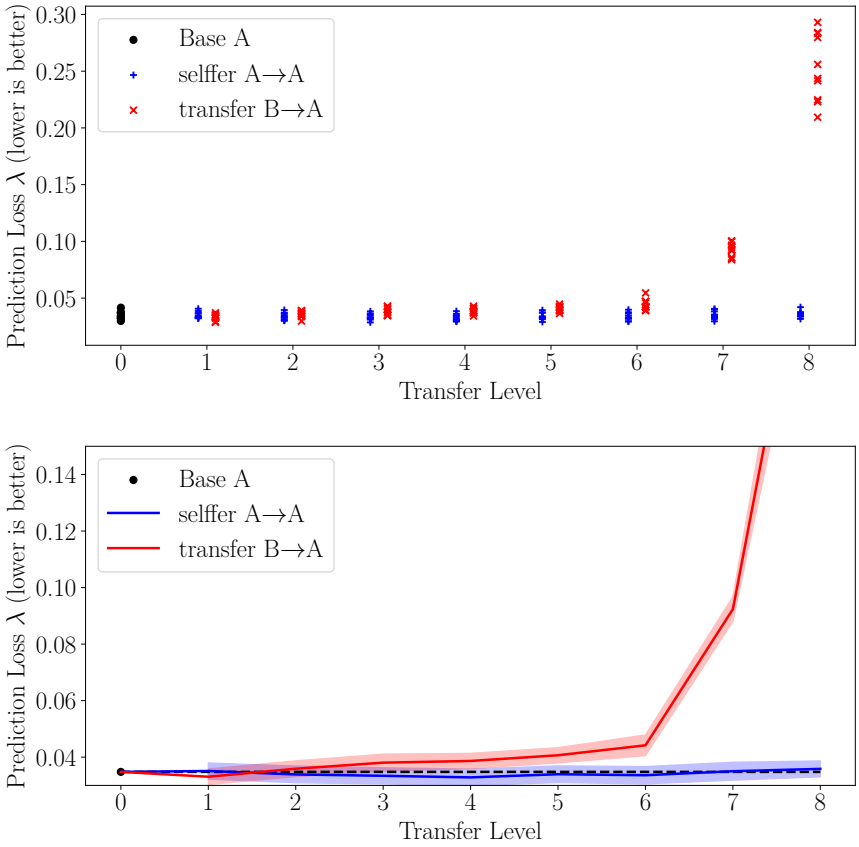


Figure 3.2: Loss plot for the Hard Feature Extraction experiment from dataset B to dataset A. The transfer level is the layer at which the base network $\mathcal{M}_S$ is chopped and frozen. See section 3.2.3 for more information on the figure generation.

3.3.2 Soft Feature Extraction

The results of the Soft Feature Extraction experiment are shown in figure 3.3. To evaluate the results, we compare the performance of the transfer and selffer models to the base

case. From the loss plots, we conclude the following:

- Again, the selffer model performance indicates that deviations of the transfer model performance from the base model must be due to the knowledge transfer. Not surprisingly, the selffer model performance never deviates significantly from the base model performance.
- The transfer model shows a slight statistically significant performance boost compared to the base model. This is in accordance with the results of Yosinski et al. [24] who also find a performance boost when transferring and not freezing layers. Hence, transferred weights in Temporal Convolutional Networks maintain the knowledge from the source domain if they are not frozen, resulting in an increased predictive performance on the target dataset.
- Unlike the Hard Feature Extraction, we no longer see a performance drop when transferring also very deep layers. Keeping weights unfrozen therefore allows layers to recover from their specificity, resulting in a relatively constant performance irrespective of the transfer level.

As observable in the transfer from dataset B to dataset A, a transfer with the Soft Feature Extraction approach may result in a performance boost. However, the reason for this performance boost is not obvious. The boost must be rooted in the difference of the two datasets with regards to information that can be regarded as general for both datasets. This difference could be either that both datasets contain a different *amount* of general information and that the general layers of base B generalize better on both domains. An alternative or concurrent effect might be that both datasets contain a different *kind* of general information that adds up when transferring from one domain to the other. To find out which of these two effects is present, we run the experiment again with switched datasets, i.e. we now transfer from dataset A to dataset B. If the previously observed performance boost is due to the difference in amount of information, we should not observe this effect when switching the datasets. On the other hand, if the effect comes from the different kind of information in the datasets, we expect to observe a similar result. If both effects are present, a performance boost should still be observable but with a different magnitude. The respective loss plot can be observed in figure 3.4. A significant performance boost can no longer be observed with the exception of one transfer level. We therefore conclude that the performance boost for the transfer from dataset B to dataset A is not primarily due to a different kind of information in the two datasets that add up. Instead, dataset B seems to contain more information that can be regarded as general and therefore generalizes well on both datasets.

3.3.3 Full Weight Initialization with partial Freeze

Results of the Full Weight Initialization with partial Freeze experiment can be observed in figure 3.5. Again, the plot is evaluated by comparing the selffer and transfer model

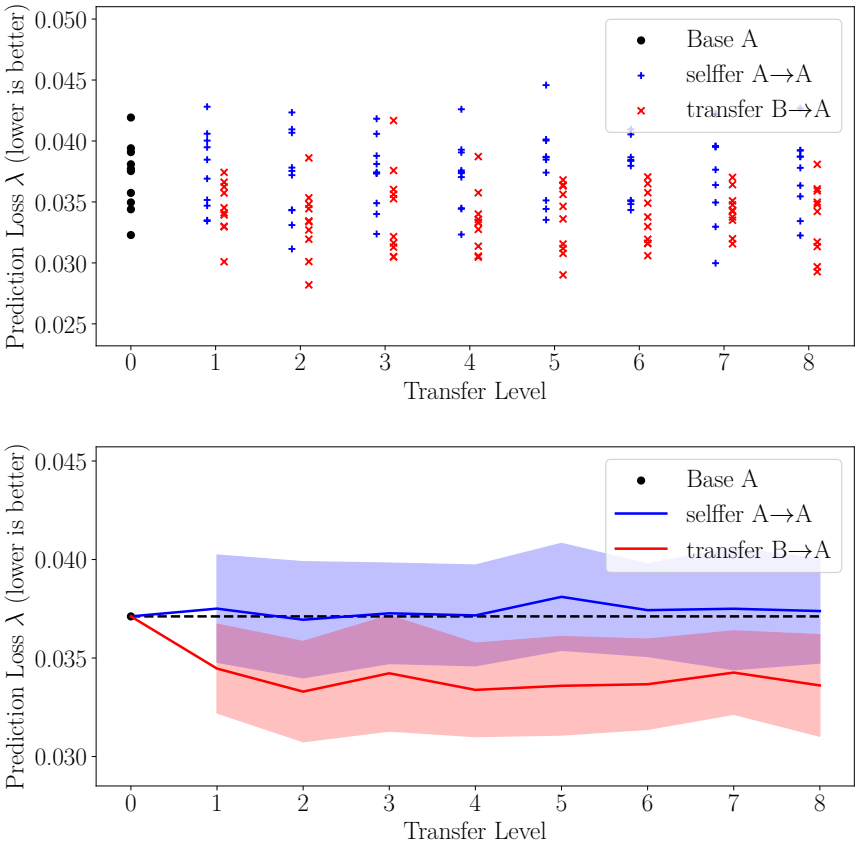


Figure 3.3: Loss plot for the Soft Feature Extraction experiment from dataset B to dataset A. The transfer level is the layer at which the base network $\mathcal{M}_S$ is chopped. No layers are frozen. See section 3.2.3 for more information on the figure generation.

performances to the base model.

The result looks similar to the one of the Hard Feature Extraction experiment. One qualitative difference to the Hard Feature Extraction is a slight performance boost for transfer level $l = 1$. This difference can be assumed to result from the previously explained difference of the datasets (cf. section 3.3.2). In fact, the small boost for $l = 1$ can no longer be observed with reversed datasets, as the interested reader may verify with the result plot in the appendix (cf. figure A.4). Because the boost can only be observed for $l = 1$, we infer that the performance boost we observed for the Soft Feature Extraction is mainly a result of transferring the first two layers and leaving these unfrozen. Once we transfer *and* freeze the first two convolutional layers or more, the performance boost is no more observable.

3.3.4 Random Reference

The idea of the reference experiment with random weights is to find out whether a performance difference of transfer models from the base model is a result of the transferred

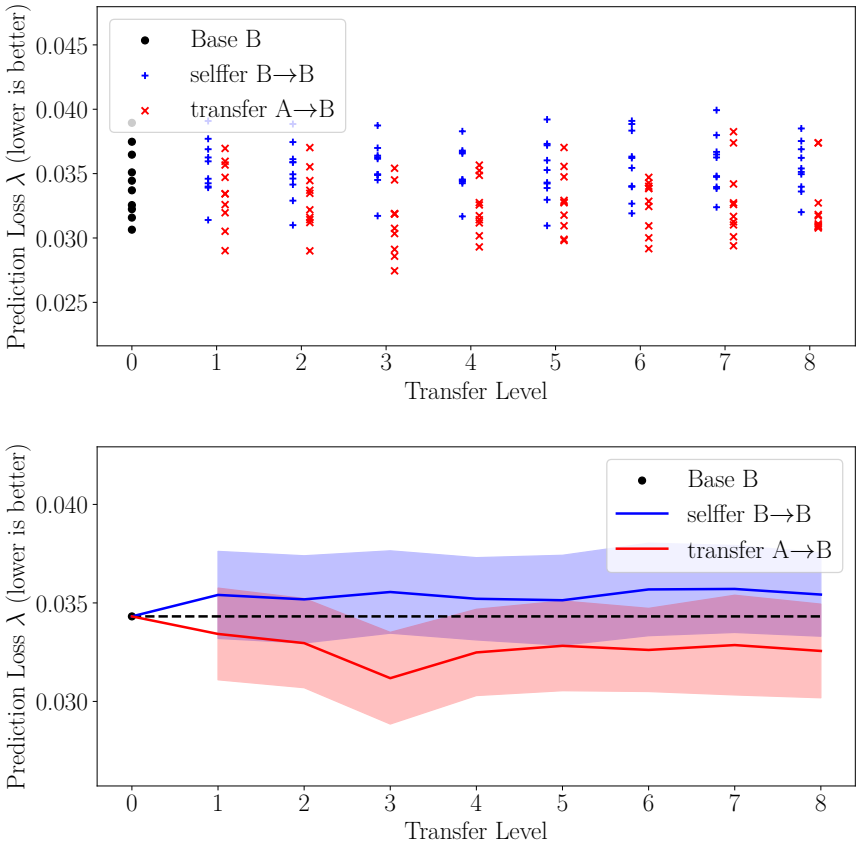


Figure 3.4: Loss plot for the Soft Feature Extraction experiment from dataset A to dataset B. The transfer level is the layer at which the base network $\mathcal{M}_S$ is chopped. Remaining layers are left unfrozen. See section 3.2.3 for more information on the figure generation.

knowledge or simply of the constrained number of degrees of freedom. The results for this random reference experiment are depicted in figure 3.6. Note that in this case, no selffer plot is shown due to the irrelevance of a transfer from random weights to a random domain. The base model in this reference experiment is a completely untrained network with randomly initialized weights.
From the results, we conclude the following points:

- The naive prediction for a normalized series would be constantly 0, resulting in a loss of 1 (equal to the standard deviation). The base model loss with an MSE close to 2 on the other hand is well above this naive prediction loss.
- At the same time, one can observe that a model with a randomly initialized and frozen first layer still results in a performance that is comparative to transfers from a pre-trained model (a plot comparing all loss curves can be found in figure 3.9 in section 3.4). This shows that for the prediction task the network architecture may be slightly oversized. The network is still able to reach a relatively good performance

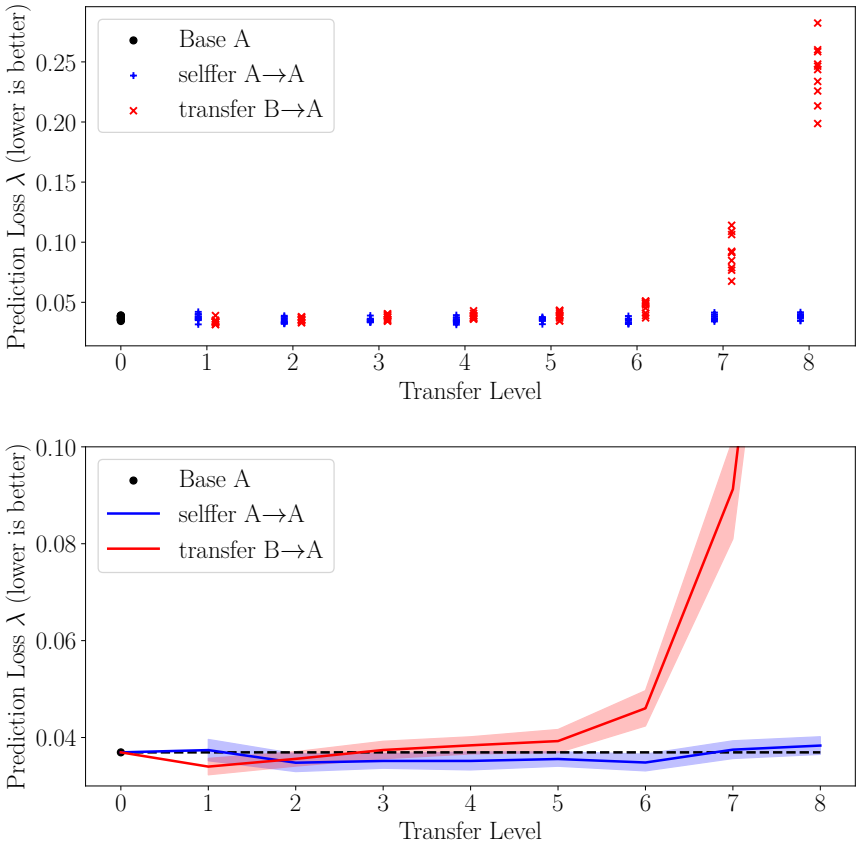


Figure 3.5: Loss plot for the Full Weight Initialization with partial Freeze from dataset B to dataset A. The transfer level is the layer at which the completely transferred base network $\mathcal{M}_S$ is frozen. See section 3.2.3 for more information on the figure generation.

with one layer less. However, since the architecture is organized in residual blocks that comprise two convolutional layers each, reducing the layer number by just one is not possible without changing the baseline of the architecture.

- For transfer levels $l > 1$, the loss becomes large compared to transferring from a pre-trained model. The slope of the loss changes comparatively much at layers 2 and 6. Strong increase of the losses for the the first and last residual block (layers 1, 2, 7 and 8) indicate a high relevance of these layers for the overall predictive power of the network. The middle of the network (layers 3 - 6) appear to be less relevant given the smaller gradient of the loss curve.

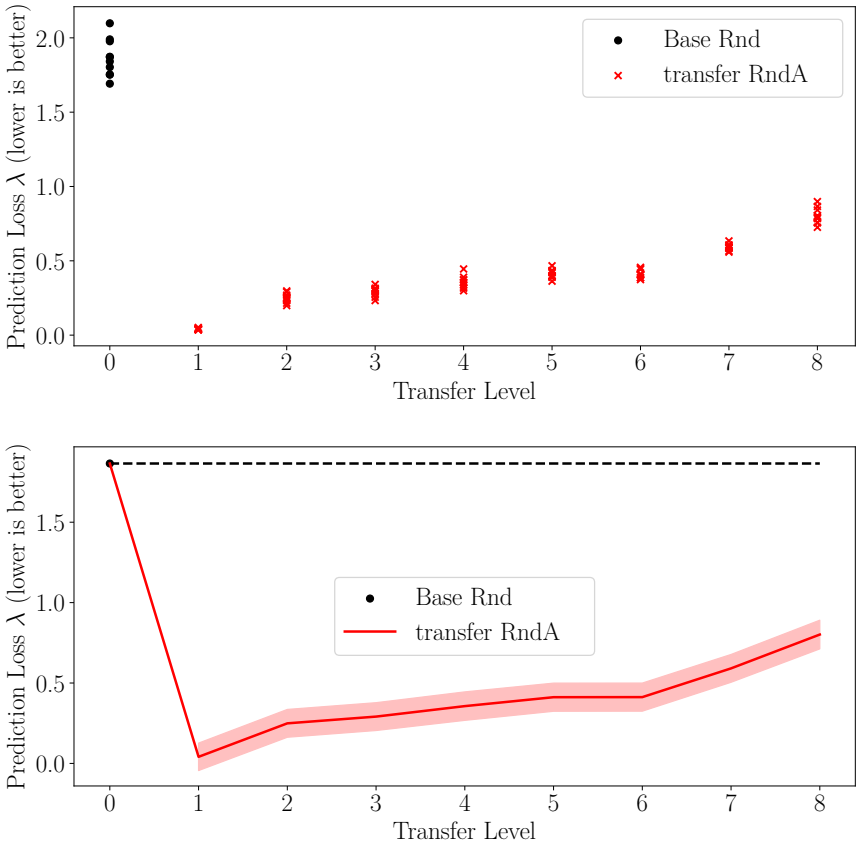


Figure 3.6: Loss plot for the random reference experiment with target dataset A. The plot shows the model losses for transfers from random weights to dataset A. The transfer level is the layer at which the random base network $\mathcal{M}_S$ is frozen. See section 3.2.3 for more information on the figure generation. The selffer plot is omitted due to the irrelevance of transferring random weights to a random domain.

3.3.5 Reduction of Dataset Knowledge

In the result sections 3.3.2 for the Soft Feature Extraction and 3.3.3 on the Full Weight Initialization with partial Freeze, we presented our findings that a dataset difference in datasets has the potential to yield a performance boost through which the transfer model achieves better predictive power than the base model. We further investigate this point by repeating all experiments with a reduced version of the previously used datasets. To reduce the amount of knowledge that the models may learn from the data, we only feed every fifth possible training sequence into the network during the training process. All other parameters are kept the same. As an example, we show the resulting plot for the Hard Feature Extraction from dataset B to dataset A in figure 3.7. The interested reader can find plots for the other experiments with this transfer direction as well as with reversed datasets in appendix A.1.
The results shown in figure 3.7 confirm the effect that we observed before with smaller

magnitude. The transfer from dataset B to dataset A yields a significant performance boost that is large compared to the one we could observe with the complete datasets. Note that figure 3.7 shows results for the Hard Feature Extraction which did not exhibit this phenomenon before. A performance boost can also be observed for the other two experiments with the same transfer direction, but not when reversing the reduced datasets, which the interested reader may verify using the figures in appendix A.1. The previous conclusion from section 3.3.2 that dataset B contains more general information than dataset A is therefore reinforced by the setup with reduced datasets.

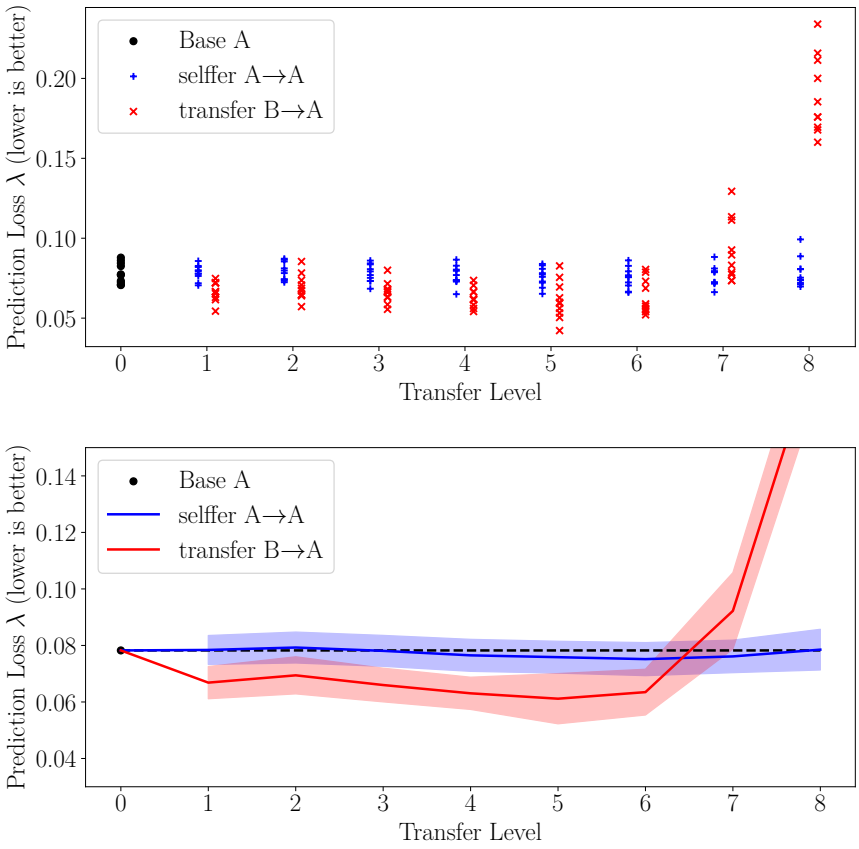


Figure 3.7: Loss plot for the Hard Feature Extraction experiment from dataset B to dataset A with reduced datasets. The transfer level is the layer at which the base network $\mathcal{M}_S$ is chopped and frozen. See section 3.2.3 for more information on the figure generation.

3.3.6 Training Duration

So far, we focused on the performance of the different selffer and transfer models in terms of MSE loss. We evaluated if knowledge in TCNs is transferable and whether there is evidence that layers transition from general to specific. Both aspects could be confirmed. Such information is particularly helpful when a model should be generated for some task that only few training data is available for. In such cases, transferring general information

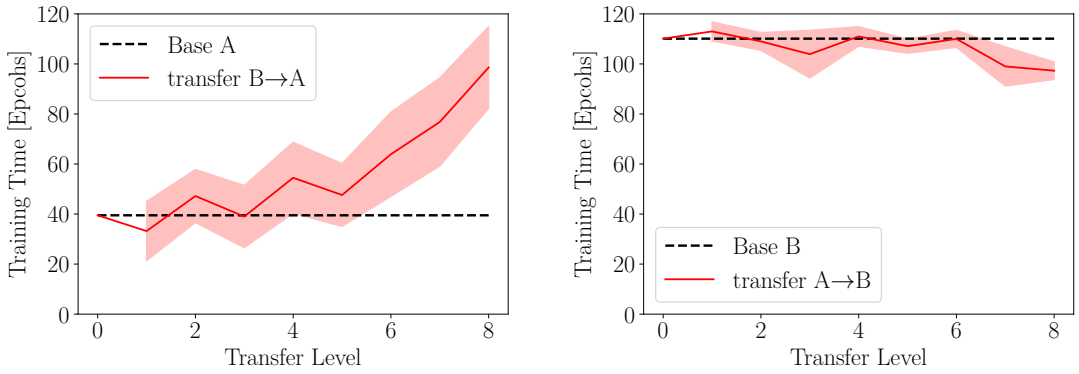
can reduce the amount of required training data. Besides this aspect, another important goal of Transfer Learning is to reduce training time. In this section, we therefore evaluate the training duration of the models during the conducted experiments in addition to the previous discussion regarding transferability.
As described in section 3.2, we do not necessarily always evaluate the model obtained after the final training epoch. Instead, we use the best model that was found after any epoch during the entire training process. By evaluating after which epoch the best model was found, we can compare if the training process was accelerated or slowed by the transfer.
Average training times for all experiments dependent on the transfer level can be observed in figure 3.8. For the three main experiments, we show the duration in epochs after which the best transfer and base models were obtained. As for the previous figures, confidence intervals around the curves indicate the acceptance region of the null hypothesis which states that the mean training duration of the transfer model and the mean training duration of the base model are equal. Note that we omit the selffer model training duration because selffer models never show a statistically significant performance difference to the base model. While this was a desired characteristic to validate the plausibility of the transfer model, the additional training time of the selffer models is (while desired) in principle wasted training time in terms of generating a model with high predictive power. For a real-world application, this training time does not make sense and is therefore not further evaluated in this section.
We conclude the following points based on the duration plots:

- Training durations with B as target dataset (right column of figure 3.8) are longer by more than factor 2 than those with target dataset A (left column). This applies in general both to the transfer models and the base models. The longer training times for dataset B can be explained by the larger amount of information that the dataset contains, as concluded in the previous sections.

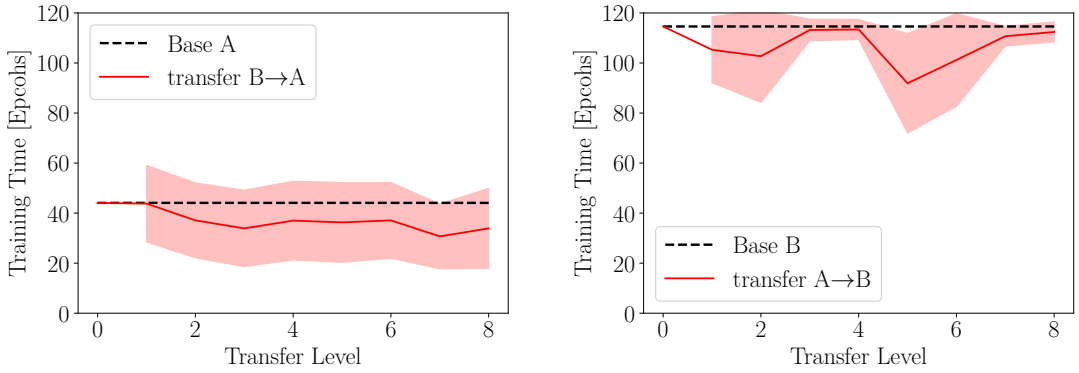
- As for the performance results, the Hard Feature Extraction and Full Weight Initialization with partial Freeze experiments show comparable results also with regards to the training duration (see figure 3.8a and 3.8c). In both cases, the transfer from dataset A to dataset B (right column of figure 3.8) shows training times comparable to the base model for most transfer levels except for the last residual block (layers 7 and 8), where the transfer model trains slightly faster than the base model. The transfer with reversed datasets from B to A (left column) on the other hand exhibits the opposite behaviour. Here, the training duration goes up with higher transfer levels. A potential explanation for this effect is that with less degrees of freedom left to train, training duration depends less on the dataset but on the architecture of the parameters to train. For example, when training the dense layer only (transfer level 8 for plots 3.8a and 3.8c), all experiments show similar durations irrespective of the direction of transfer. The more degrees of freedom (hence, for smaller lower transfer levels), the training duration appears to depend more on the dataset. In these cases, the transfer model training time is similar to the duration for the base model.

- In case of the Soft Feature Extraction (figure 3.8b), the training duration of the transfer model shows almost no significant deviation from the training time of the base model. This corresponds to the explanation for the previous point. In the case of the Soft Feature Extraction, the number of degrees of freedom is not constrained and training durations are similar to the base model.

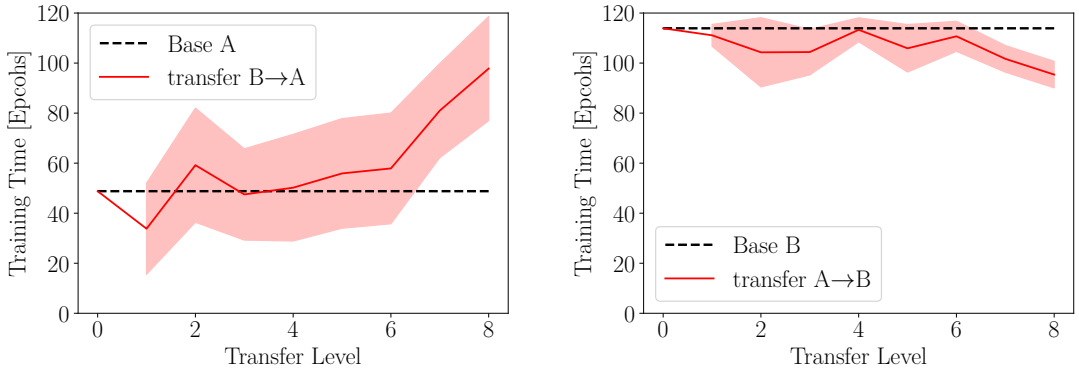
- We generally observe no significantly reduced training time compared to the base model except for transfer levels $l > 6$ of the Hard Feature Extraction and Full Weight Initialization with Partial freeze experiments. However, the previous experiments have shown that these scenarios result in a low performance of the model. Hence, a positive effect on the training duration resulting from a knowledge transfer cannot be identified. At the same time, the mostly insignificant deviations from the base model for $l \leq 6$ also does not hint to a disadvantage in terms of training time. The low transfer levels are particularly interesting since these are the ones that still achieve a good predictive power.



(a) Hard Feature Extraction



(b) Soft Feature Extraction



(c) Full Weight Initialization with partial Freeze

Figure 3.8: Duration plots for the three main experiments (a) Hard Feature Extraction, (b) Soft Feature Extraction and (c) Full Weight Initialization with partial Freeze. Plots show the average training duration in epochs that were needed to find the best model at a given transfer level. Dashed lines mark the mean duration to train the base model. Red lines show the training duration for transfer models. Confidence intervals indicate whether the transfer training time shows statistically significant difference to the base model training duration. Selffer model training times are omitted because throughout all experiments, the selffer model never shows significant performance difference to the base model. *left column:* Transfer from dataset B to dataset A. *right column:* Transfer from dataset A to dataset B.

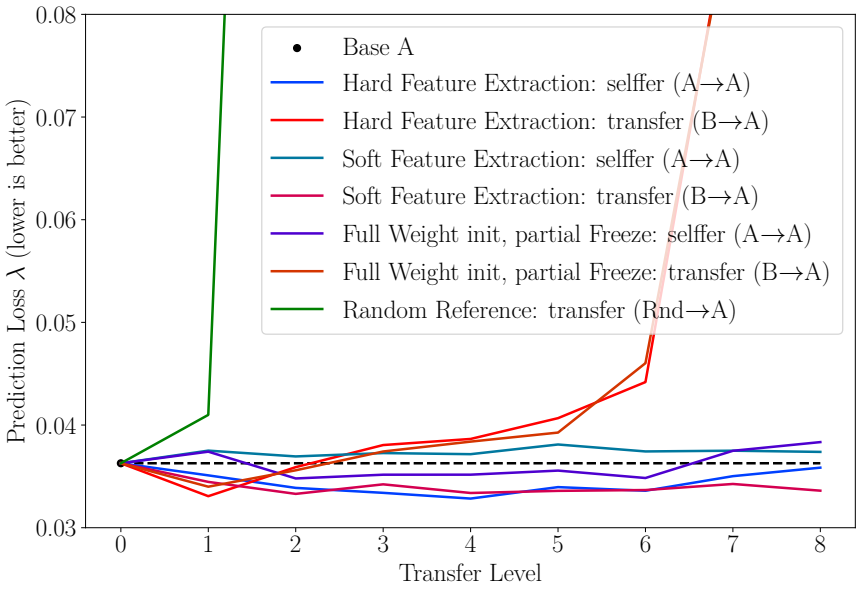
3.4 Discussion

In the previous section 3.3, we presented the results of the three main experiments, the reference experiment, an evaluation of training duration and a reduction of dataset knowledge separately. We now bring these results together and discuss the overall findings that result from the research on layer transferability that was presented throughout this chapter. Figure 3.9 shows the results of all experiments for both the transfer from dataset B to A and from dataset A to B (using the complete datasets). Besides the transfer and selffer model performances of the three main experiments, the result of the random reference experiment is shown.
As one can see, transfer models (figure 3.9, lines in red tones) always perform better than models with random frozen weights (green line). This shows that knowledge is actually transferred between domains and the transfer model does not simply use remaining degrees of freedom in unfrozen layers to learn knowledge from the target domain. The difference between the main experiments with layer freezing and the random experiment can be considered a metric for the transferred knowledge.
The comparison highlights again that there is no qualitative difference between the Hard Feature Extraction and the Full Weight Initialization with partial Freeze. Initializing deep layers with random weights seems just as good as transferring these layers and leaving them unfrozen. Furthermore, we find no fragile co-adaptation. Selffer models show similar performance as the base model in all three experiments. Based on these two findings, we can simplify figure 3.9 by leaving out the selffer model curves and using the Hard Feature Extraction curve as a representation also for the Full Weight Initialization with partial Freeze. The resulting figure 3.10 reveals three areas that we interpret as follows:

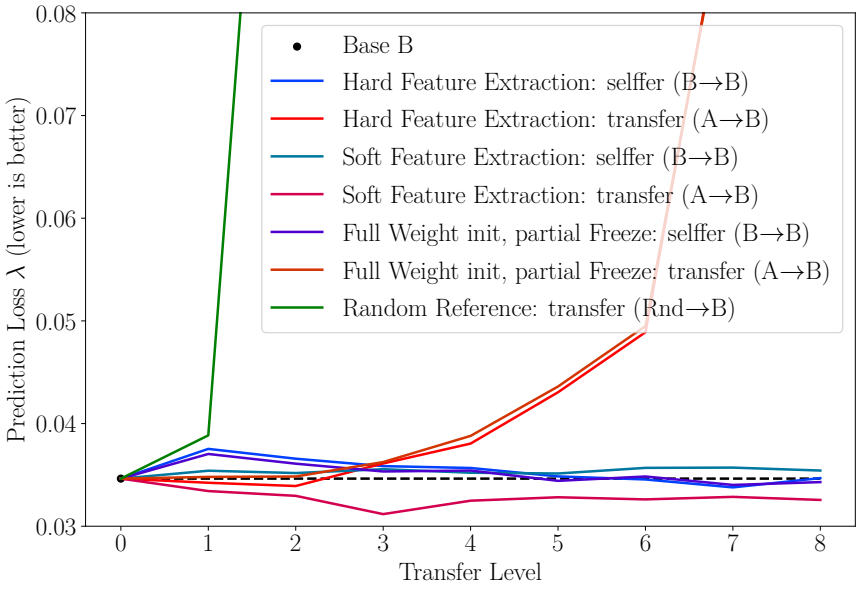
A: Dataset Difference. As discussed in the result section for the Soft Feature Extraction, we observe a performance difference of the transfer model that results from the different amount of general information in the datasets of the two domains. We find the same effect for the two other experiments when reducing the amount of information that is contained in the datasets. The Soft Feature Extraction also shows that unfreezing layers does not cause layers to unlearn the transferred knowledge. The respective performance deviation to the base model is marked in figures 3.10a and 3.10b as region A between the line of the Soft Feature Extraction experiment and the base model loss.

B: Specificity. The Hard Feature Extraction and Full Weight Initialization with partial Freeze reveal that layers transition from general to specific, resulting in an increase of the loss the more layers are transferred and frozen. With the Soft Feature Extraction, we also showed that layers recover from specificity when left unfrozen. The specificity effect is marked in figures 3.10a and 3.10b as region B between the Soft Feature Extraction and the base model.

C: Knowledge Transfer. Finally, the random reference model shows that transferred weights always perform better than randomly initialized layers. This allows the



(a) Transfer to dataset A



(b) Transfer to dataset B

Figure 3.9: Loss plot of all main experiments and the reference experiment.

conclusion that transferring knowledge between the two domains actually yields an advantage in terms of prediction loss. The transferred knowledge can be quantified by the performance gain that frozen layers of the Hard Feature Extraction exhibit compared to completely random weights. The resulting area C in figures 3.10a and 3.10b is the one between the curves of the Hard Feature Extraction and the random reference.

Through our experiments for the evaluation of layer transferability, we were able to show

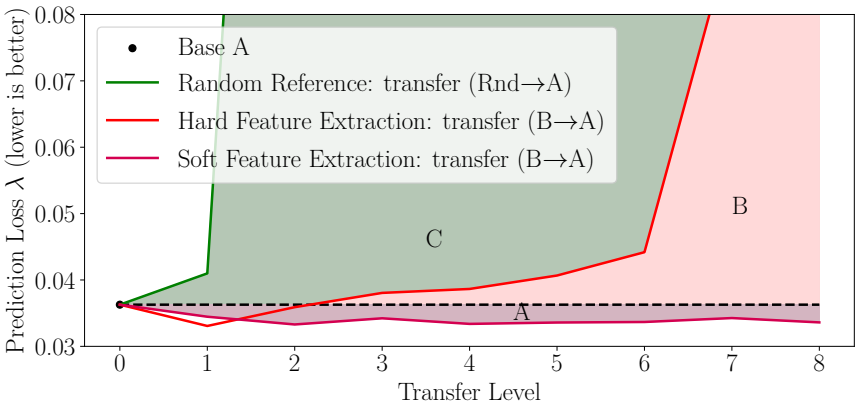
that as for regular CNNs, layer transfer among TCN models is possible. In particular, we found three effects which express as areas between the loss curves: (1) the different amount of general information in the training datasets, (2) the specificity of transferred layers and (3) the knowledge contained in the transferred weights. On the other hand, we did not observe fragile co-adaptation of layers which may be due to the particular architecture of the TCN which uses residual blocks. The successful transfer shows that low layers learn general information and the performance decrease when transferring and freezing deep layers suggests that there is a transition from general to specific features.
The evaluation of the training durations shows that a knowledge transfer can have an effect on the time that is needed to obtain a model for the target domain and task. We observed that the duration for the transfer model is typically at the same level as the training time of the base model. The training time therefore generally depends on the target dataset that the transfer model is trained on. However, the more of the transferred layers we freeze in the transfer model, the more the training seems to be determined only by the network architecture and not the dataset. This is the case only for scenarios with weight freezing and with high transfer levels which generally show poor performance and therefore are not advisable.
Based on the results of the qualitative transferability evaluation presented in this chapter, we derive suggestions for how a transfer between TCN models should be designed. We have seen that all three approaches - Hard Feature Extraction, Soft Feature Extraction and Full Weight Initialization with partial Freeze - perform equally well up to some transfer level that represents the threshold where the network transitions from general to specific. Hard Feature Extraction and Full Weight Initialization with partial Freeze have the advantage that some layers are frozen which effectively reduces the number of trainable parameters. We assume a reduction of parameters to reduce the likelihood of overfitting, although this has not been investigated throughout our experiments. Moreover, fewer parameters result in shorter training durations (see table 3.2). The Soft Feature Extraction on the other hand does not decrease in performance for transfer levels beyond the network threshold. At the same time, this approach also does not reduce the number of trainable parameters. We therefore suggest to design a transfer as follows:

1. If the transition threshold from general to specific of the source model is known, the Full Weight Initialization with partial Freeze should be chosen. This effectively reduces the number of parameters to train while achieving good transfer performance. However, if the threshold is assumed wrong, this may result in poor predictive power of the transfer model, i.e. a negative transfer. Equivalently, the Hard Feature Extraction can also be used although we suggest to always prefer the Full Weight Initialization with partial Freeze because we know that specific transferred layers still perform better than completely random layers.

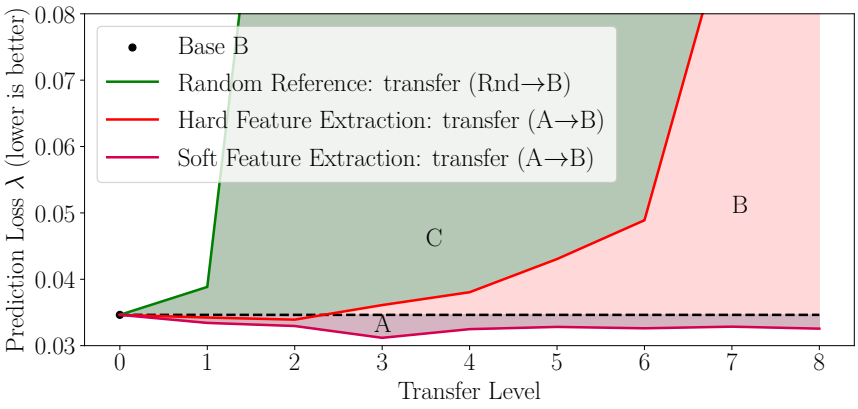
2. If the transition threshold is unknown, all layers should be transferred without freezing. This is equivalent to performing a Soft Feature Extraction with $l = L$ or a Full Weight Initialization with partial Freeze with $l = 0$. The result is a good transfer

performance and training time but no benefits in terms of number of parameters to train.

The presented qualitative results of this chapter help understand the behaviour of TCNs and give insights into how a transfer among TCN models should be conducted. However, they leave the question of what kind of information is actually learned unanswered. This aspect is covered in the following chapter that aims to understand what knowledge is contained in TCN layers by visualizing them.



(a) Transfer to dataset A



(b) Transfer to dataset B

Figure 3.10: Loss plot of the Hard Feature Extraction, Soft Feature Extraction and random reference experiment transfer models with marked areas. *(top)* Plot for the transfer from dataset B to dataset A. *(bottom)* Plot for the transfer from dataset B to dataset A. Regions between the curves result from the following effects: (A) Dataset difference, (B) Layer Specificity and (C) Transferred knowledge.

Chapter 4

Knowledge Visualization

We saw in the previous chapter that layers in Temporal Convolutional Networks transition from general to specific the deeper they reside within the network and that general knowledge is transferable between TCN models. While an acceleration of the training process cannot be generally observed, we could show that Transfer Learning bears the potential to reduce the number of training parameters for TCNs and may under certain conditions result in a performance boost. However, these qualitative findings do not give insight into the kind of knowledge that is encoded within TCN layers. In order to answer the second research question regarding the kind of knowledge that TCN layers learn, this chapter deals with the visualization of layers.
As a metric, we use the activation of layers and neurons. This concept is introduced in the first section 4.1 of this chapter. As we show in section 4.2, directly plotting the activation of convolutional layers inside residual blocks gives an indication to what task different neurons perform in the network. However, for many of the neurons, an interpretation of the visual result is not obvious. In addition to this visual evaluation, we therefore also quantify the activation levels of layers and neurons. The horizontal evaluation in section 4.3 looks into the influence that individual residual blocks have on the activation of the network. The vertical evaluation presented in section 4.4 on the other hand aims at finding the most important channels for the model prediction. Throughout all evaluations in this chapter, we used a model with parameters as described in section 3.2.1 trained on dataset B from the previous chapter. The interested reader may find results for a model trained on dataset A in appendix B and confirm that they are qualitatively similar.

4.1 Layer Activation

Throughout this chapter, we focus on the activation of layers and residual blocks in the network. In this section, we define the notion of activation for our TCN architecture and describe which information is visualized. Each residual block contains a parallel path to map the block input to its output and thus performs a modification on the signal rather than a transformation. We want to understand what kind of modifications the block performs on the input. Therefore, we visualize the signal that each block adds to the final output. For this purpose, we use hooks to obtain the intermediate output of modules in

the network[1]. The placement of these hooks is shown in figure 4.1. Hooks for residual block modifications are shown in green, while the final output after all residual blocks is marked in red.

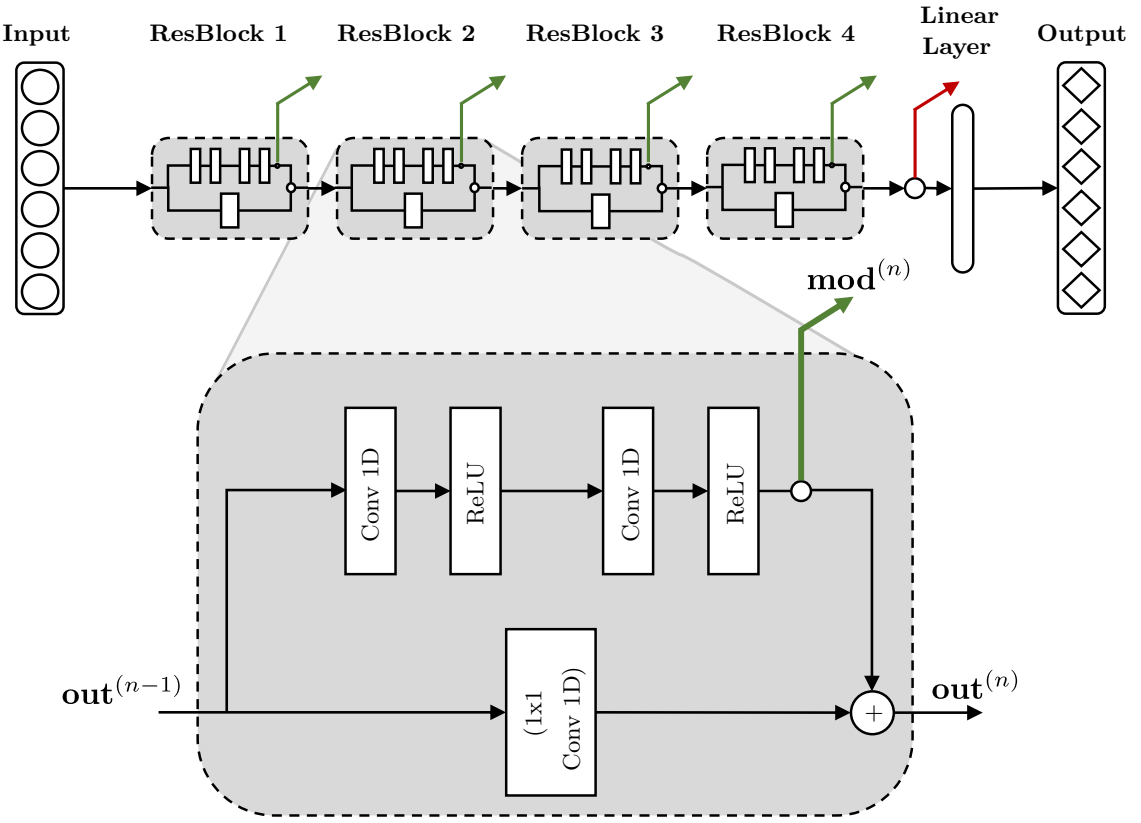


Figure 4.1: Location of hooks in the TCN for the visualization of layer activations. Hooks marked in green are used to obtain the activation of the modification that a residual block imposes on the channels. The marked red hook obtains activations of the output channels that are fed into the dense layer. The zoom shows a detailed view of the hook location within one exemplary residual block.

Each convolutional layer outputs fifty channels or signals, in accordance with the number of neurons defined in section 3.2.1. Therefore, for any input signal, we obtain fifty activation signals at each of the four green hooks, which we call the *modification* to the block input. The red hook behind the final residual block on the other hand fetches the *outputs* of this residual block. In the following, we use bold letters for vectors while regular letters are used for a single value. We call $\mathbf{mod}^{i,n}(\mathbf{x})$ and $\mathbf{out}^{i,n}(\mathbf{x})$ respectively the modification activation and output activation of channel i in block n for input sequence $\mathbf{x}$. Note that both activations are vectors of the same length T, with T the length of the input signal $\mathbf{x}$. For some arbitrary activation $\mathbf{a}^{i,n}$, we denote $a_t^{i,n}$ the activation value of channel i in block n at time step t. We calculate the overall activation $a^{i,n}$ of a channel as the sum of

[1] we use a forward hook function that can be registered on any PyTorch module of a network. This function is called every time the module computes an output and returns this output to be observed or modified. See `https://pytorch.org/docs/master/nn.html#torch.nn.Module.register_forward_hook`

the intermediate activation vector:

$$a^{i,n}(\mathbf{x}) = \sum_{t=1}^{T} a_t^{i,n}(\mathbf{x}) \tag{4.1}$$

Since the output of the final residual block is fed into a dense layer, channels that have a high activation according to (4.1) can be expected to have a bigger impact on the prediction task of the network.
Respectively, we define the activation metric of one residual block $a^{\cdot,n}$ as the sum of all channel activations within this residual block. Note the $\cdot$ in the superscript to denote the index over which is aggregated:

$$a^{\cdot,n}(\mathbf{x}) = \sum_{i=1}^{I} a^{i,n}(\mathbf{x}) = \sum_{i=1}^{I} \sum_{t=1}^{T} a_t^{i,n}(\mathbf{x}) \tag{4.2}$$

In addition, we define a^{in} as the sum over the original input signal:

$$a^{in}(\mathbf{x}) \coloneqq \sum_{t=0}^{T} x_t \tag{4.3}$$

In the same way, we define a^{out} as the sum over the output that is obtained from the hook behind the last residual block N as marked in red in figure 4.1:

$$a^{out}(\mathbf{x}) \coloneqq out^{\cdot,N}(\mathbf{x}) = \sum_{i=1}^{I} \sum_{t=1}^{T} a_t^{i,N}(\mathbf{x}) \tag{4.4}$$

4.2 Visualizing Layer Modification

As a first step to understand the kind of knowledge that is learned in the TCN and also where in the network this knowledge resides, we visualize the modification of different layers in the network. These modifications are the activations at the points in the network marked with green hooks in figure 4.1. All evaluations in this section are done for eight different input signals. These are

- the Dirac delta function (δ function),
- a sawtooth wave,
- a sine wave,
- a square wave,
- a triangular wave
- a constant input,

- a linear input and
- an input sequence of the test set from the original synthetic data.

For all eight signals, we show the channel with maximal modification activation of all residual blocks in figure 4.2 as computed by equation (4.1). Complete plots of the modification activations $\mathbf{mod}^{i,n}$ for the eight signals with all fifty channels and for all residual blocks can be found in appendix B.1.1 and B.1.2. As one can see from the maximally activated channel plots and the full plots, a direct interpretation of all intermediate activations is not always obvious. However, in the plots we find some indications for tasks that the neurons perform on the input signal. For example, for some of the signals, the maximally activated neuron in the first residual block seems to apply smoothing on the input signal (e.g. figure 4.2f). Note that it is not necessarily always the same neuron that shows the maximal activation for all of the signals. Other neurons seem to filter out frequencies of the signal, as one may assume from the maximal activation in the first residual block to the sample from the test set (cf. figure 4.2h). Again different neurons invert or shift the input signal, like the maximally activated neuron in the third residual block in case of the sine input signal (cf. figure 4.2e). More such examples can be found in the plots of all neurons for all signals in the appendix. However, this visual interpretation of the activations remains relatively subjective and demands for a quantifiable metric. Therefore, besides looking at plots of intermediate activations on a time scale, we also use the activation values as a metric to see which neurons and layers exhibit the most influence on the output of the network. In the following section, we perform a horizontal evaluation in which we aim to quantify the influence of each layer on the final output.

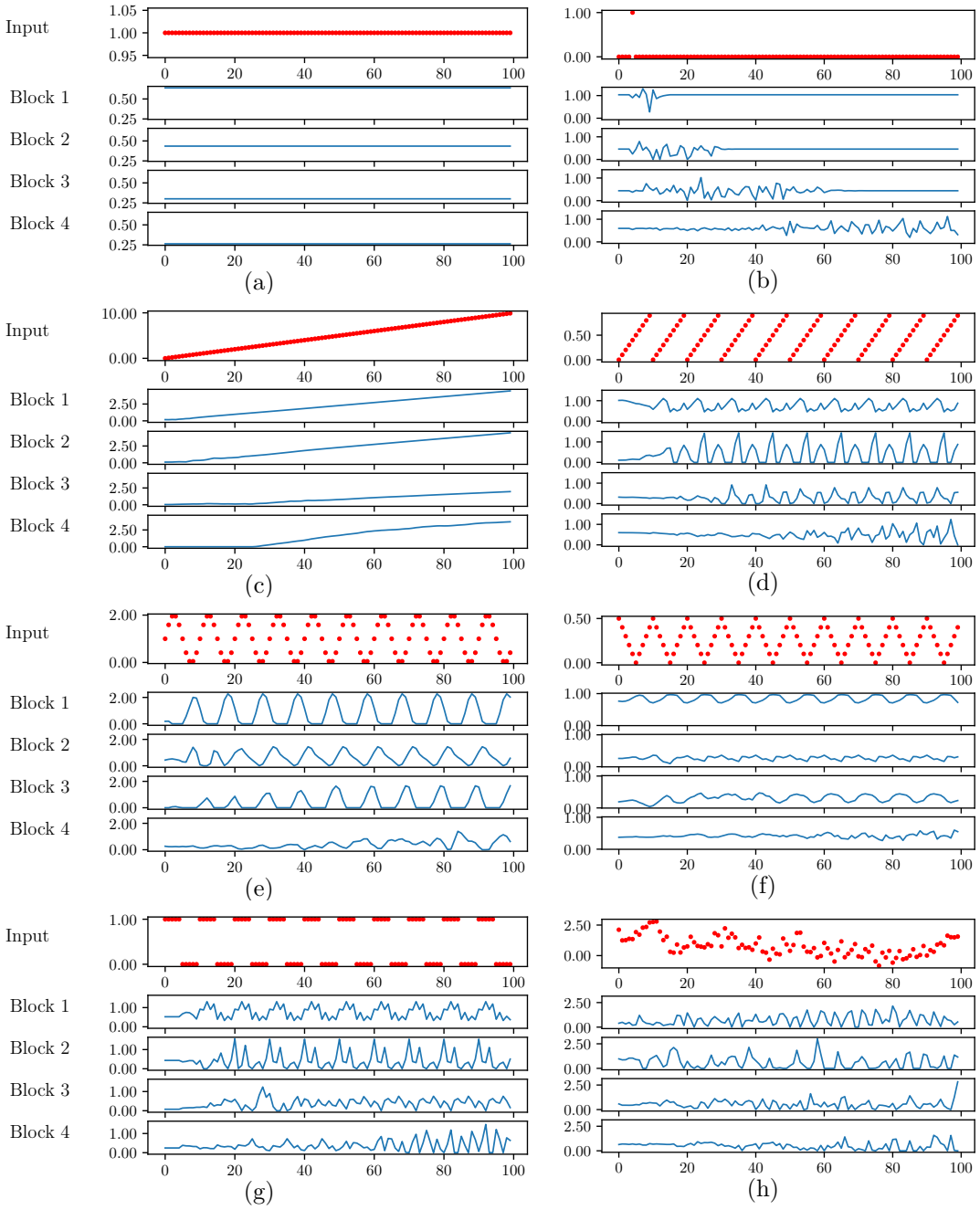


Figure 4.2: Activation of the maximally activated neuron in each layer for eight different input signals. Signals are chopped at $t = 100$ for better readability. Plots in red show the signals used to stimulate the network. Blue plots are the modifications that the maximally activated neurons of each residual block add to the channel. Input signals are (a) the constant signal 1, (b) a dirac function, (c) a linear function with slope 0.1, (d) a sawtooth with amplitude 1, (e) a sine with offset 1, (f) a triangular wave with amplitude 0.5, (g) a square wave with amplitude 1 and (h) a sample drawn randomly from the test set of the dataset. All periodic signals have period length 10.

4.3 Horizontal Evaluation

As we have seen before, directly interpreting the temporal plots of intermediate activations is not always obvious. With the horizontal evaluation, we aim to find out which of the residual blocks have the strongest influence on the final prediction result. We measure this influence in terms of activation that residual blocks add to each channel in the network. Since a residual block performs its modification to the input signal in an additive manner, the influence of one residual block can be easily expressed as the fraction of the final activation of the model that was added by this specific residual block. We calculate the influence i_n of a residual block n with regards to an input signal $\mathbf{x}$ as follows:

$$i_n(n, \mathbf{x}) = \frac{mod^{\cdot,n}(\mathbf{x})}{a^{out}(\mathbf{x}) - a^{in}(\mathbf{x}) * I} \tag{4.5}$$

The denominator in the equation is the absolute modification that the model adds to all channels as per equation (4.2) and the nominator is the activation of the modification from residual block n, using equations (4.3) and (4.4). Note the multiplication of the input signal activation a^{in} with the number of channels I to account for the (1×1) convolution in the first residual block. For only positive input signals, the influence values of all residual blocks in the network sum up to 1. This may not be the case for input signals with negative values due to the additional ReLU nonlinearity *behind* each residual block (see paragraph *Architecture* in section 2.1.2). Because of this nonlinearity, some of the overall modification to the input signal may result from the additional ReLU and not from one of the modifications *inside* a residual block. For this reason, all input signals are chosen to be non-negative with the exception of the samples from the test set.
The resulting influences of the residual blocks are displayed in figure 4.3 for the aforementioned eight different signals. Note that in this case we do not only use a single sequence from the test dataset but calculate the average influence for all available test sequences. With a total length of 2000 samples in the test set and an input sequence length of 200 samples, this results in a set of 1800 input sequences in the test set X_{test} and the same number of activation vectors per hook. The mean influence $\bar{i}_n(n)$ of all 1800 test sequences is calculated as follows, using (4.5):

$$\bar{i}_n(n, X_{test}) = \frac{1}{|X_{test}|} \sum_{\mathbf{x} \in X_{test}} i_n(n, \mathbf{x}) \tag{4.6}$$

The results show that for all signals, the first residual block has the biggest influence on the output activation with 55.5% on average. The third and fourth residual blocks have the lowest influence with on average 12.7% and 12.8%. With 17.9% influence, the second residual block exhibits a higher influence than the last two blocks while also showing higher variance depending on the input signal. All influence values are tabulated in table 4.1 together with the mean, minimal and maximal values. Note that the results from figure 4.3 match our findings from the previous chapter. The investigation into the transferability

of layers suggests that the importance of layers decreases with the depth in the network. This is particularly visible for the average of all sequences in the test dataset (grey line in figure 4.3). For these test set sequences, the influence decreases monotonously with the depth of the residual block in the network.

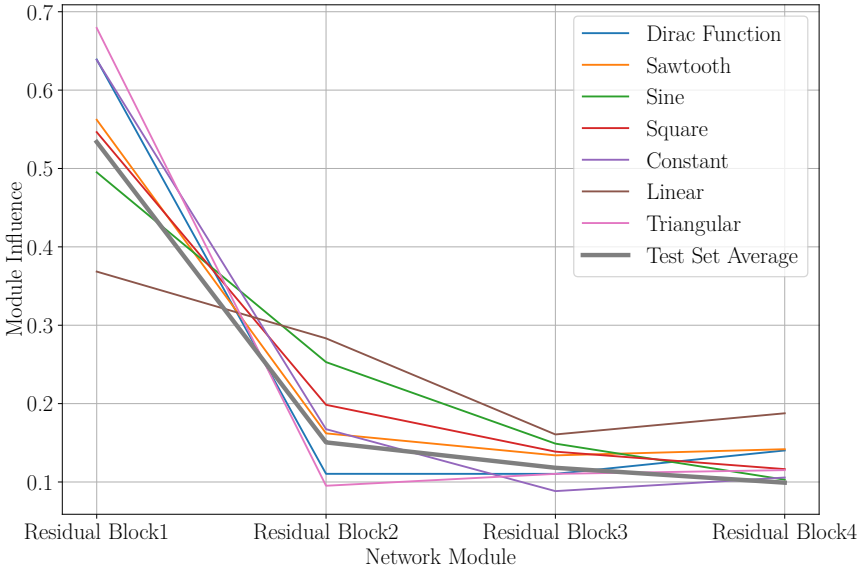


Figure 4.3: Influence of residual blocks on the total activation of the network trained on dataset B. All modification curves except the *Test Set Average* result from a single input sequence of length 200. *Test Set Average* is the average influence of all sequences in the test set of B. The respective values can be observed in table 4.1.

	ResBlock 1	ResBlock 2	ResBlock 3	ResBlock 4
Dirac Function	63.9	11.0	11.0	14.0
Sawtooth	56.2	16.2	13.4	14.2
Sine	49.5	25.3	14.9	10.3
Square	54.6	19.8	13.9	11.6
Constant	63.9	16.7	8.8	10.6
Linear	36.8	28.3	16.1	18.8
Triangular	67.9	9.5	11.0	11.5
Test Set Average	53.4	15.1	11.8	9.9
Mean	55.5	17.9	12.7	12.8
Min	36.8	9.5	8.8	9.9
Max	67.9	28.3	16.1	18.8

Table 4.1: Layer influence for different input signals of a TCN trained on dataset B. Influence values are calculated according to equation (4.5). *Test Set Average* denotes the average of all sequences in the test set of B. All values in percent of the output activation of the final residual block.

4.4 Vertical Evaluation

As presented in the previous section, layers have a different influence on the final output of the network. We showed that the first residual block has the strongest impact and that the influence decreases with the depth of the block within the network. The influence was measured as the sum of the modification activation on all channels that the block adds to the output. Similarly, we now want to find out which of the channels are most important. For this evaluation, we do not use the eight standard signals as before. Instead, we use the activations resulting from all possible input sequences in the test set from dataset B as in the previous section. For all 1800 test sequences in the training set X_{test}, we calculate the mean activation sum $\bar{a}^{i,n}$ using (4.1) for each channel:

$$\bar{a}^{i,n}(X_{test}) = \frac{1}{|X_{test}|} \sum_{\mathbf{x} \in X_{test}} a^{i,n}(\mathbf{x}) \tag{4.7}$$

To evaluate the importance of channels in the final output, we look at the activation of all fifty channels after the last residual block, i.e. the activations $\mathbf{out}^{i,4}(\mathbf{x})$ for all channels i. This corresponds to the activation obtained from the hook marked in red in figure 4.1. We interpret the activation of a channel as its importance for the prediction of the network. However, as one can see in the figure, this activation is still an intermediate result that will be processed by the final dense layer of the network. To also account for the weighting of this dense layer, we multiply each of the channel activations with their respective weight in the dense layer. The resulting activations are shown, ordered by magnitude, in figure 4.4. Clearly, channel 32 has the largest influence. The influence of channel 11 is smallest and has almost no impact on the output signal[2]. Since channels with a high activation have a bigger impact on the output of the model, we may assume that these are also channels that contain a larger amount of knowledge learned by the model. One can also see that the channel activation decreases only moderately with a slight tendency towards a heavy-tail distribution.

Another interesting aspect that can be observed in table 4.2 is the number of dead modifications, i.e. a residual block that does not modify a channel. This happens when the neuron related to some channel inside the second convolutional layer of a residual block is not activated. The number of dead modifications is displayed in table 4.2 for two models, trained on dataset A and B and evaluated on the respective dataset. We evaluate dead modifications for all test sequences in the datasets. For example, there are four channels in the model trained on dataset A that are never altered by the first residual block given any of the test sequences from A. The number of dead modifications is highest for the last residual block. We have seen in the previous horizontal evaluation that deeper residual blocks are less activated and thus contain less knowledge than the first one. The increasing number of dead channels is therefore not surprising and underlines the fact that while deep layers are more specific, they also contain in general less knowledge. Interestingly,

[2]Note that the channel numbers have no meaning and the channel numbers corresponding to the neurons with the highest activation are simply a result of the random weight initialization of the model.

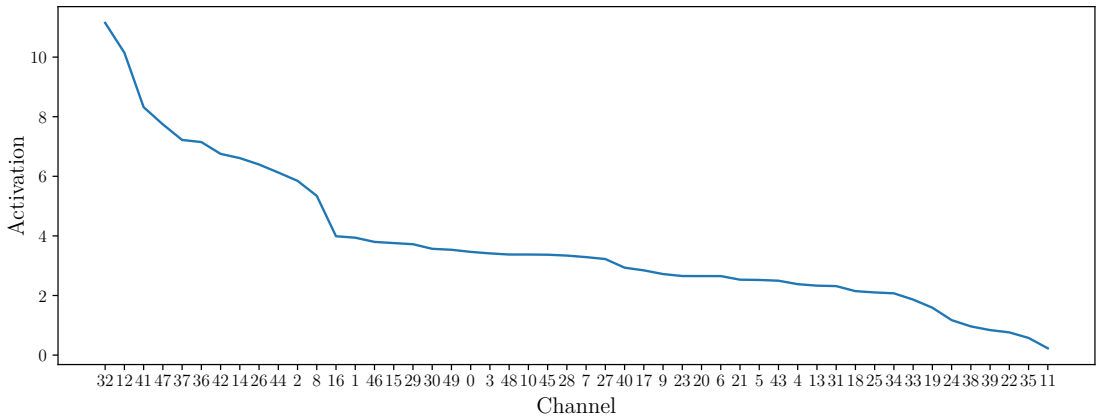


Figure 4.4: Activation distribution of all fifty channels of the final network output. Activation values are the activations after the last residual block, scaled by the respective channel factors of the final dense layer of the network.

this tendency is stronger for the model trained on A than for the model trained on B. The higher number of dead modifications in deep layers of the model trained on A suggests that the model trained on B contains more specific information than the model trained on A.

Table 4.2: Number and percentage of dead modifications in each residual block for two models trained on dataset A and dataset B.

	ResBlock 1	**ResBlock 2**	**ResBlock 3**	**ResBlock 4**
Model A	4 (8%)	2 (4%)	7 (14%)	13 (26%)
Model B	6 (12%)	3 (6%)	4 (8%)	9 (18%)

4.5 Discussion

In this chapter, we focused on a visualization of layers and neurons within TCNs. We started with a visualization of intermediate layer activations which indicates that some neurons in a TCN perform tasks like smoothing, filtering, shifting and inverting. The visualized activations are the modifications that residual blocks add to each channel. We highlight that the particularity of skip connections within residual blocks needs to be taken into account when interpreting the visualization of modifications. Due to these skip connections, the activation cannot be considered a mere result of combining and weighting features from the previous layer. Instead, activations depend both on the features from all previous layers *and* the original input. This makes a visual interpretation less intuitive.
In a horizontal evaluation, we quantified and visualized the influence that residual blocks have on the total activation of the network output. The results show that the first residual block has the highest influence and accounts on average for more than 50% of the total activation of the network. Evaluated on the sequences from the test set, the influence of

residual blocks drops monotonously with their depth in the network. This suggests that while very specific, deep layers contain less information.
We finally performed a vertical evaluation to find the most important channels inside the network. We found that the activation distribution exhibits a tendency towards a heavy tail but with comparatively moderate slope. The evaluation of dead modifications shows that deep residual blocks tend to produce more dead modifications, which was expected due to their specificity.

Chapter 5

Quantification of Layer Knowledge

In the previous chapters, we have shown which layers in TCNs are well transferable and that layers transition from general to specific the deeper they are located in the network. We also analyzed individual layers and channels to better understand what kind of knowledge is encoded in layers of the TCN. After having shown *that* specific and general knowledge is learned in TCNs and *what* this knowledge is. To answer the third research question, we now want to quantify *how much* knowledge the different layers contain. For this purpose, we conduct an experiment that is inspired by the experiments for the qualitative evaluation of layer transferability in chapter 3. Again, we evaluate models that are generated using a certain number of pre-trained layers from a base network. However, unlike in the experiments conducted in chapter 3, we now map the transferred layers directly to the dense layer of the TCN and evaluate the performance of the resulting reduced model. The performance difference that results from adding one layer is used as a metric for the amount of knowledge that this layer contains.

5.1 Experiment Overview

As for the experiments in chapter 3, we train two base models $\mathcal{M}_T$ and $\mathcal{M}_S$ on a target dataset from domain $\mathcal{D}_T$ and a source dataset from a source domain $\mathcal{D}_S$. We use the same datasets A and B that we used throughout the previous experiments. As before, the experiment is repeated ten times to reduce the influence of outliers. We also run the experiment in both directions, with A as source dataset and B as target dataset and vice versa. The following steps are carried out for each of the ten repetitions:

1. **Training of two base models $\mathcal{M}_T$ and $\mathcal{M}_S$ on the two datasets.** Models are generated with the same hyperparameters as the base models for the qualitative transferability evaluation (cf. section 3.2.1). The best model during the entire training process of 120 epochs is selected as final base model.

2. **Generation of two reduced models $\mathcal{M}_T^*$ and $\mathcal{M}_S^*$.** From the base models, we derive two new models $\mathcal{M}_T^*$ and $\mathcal{M}_S^*$ as visualized in figure 5.1. These models are generated by keeping only the first l convolutional layers of the based models, with $l \in [0; L-1]$. In other words, the new models are created by dropping the

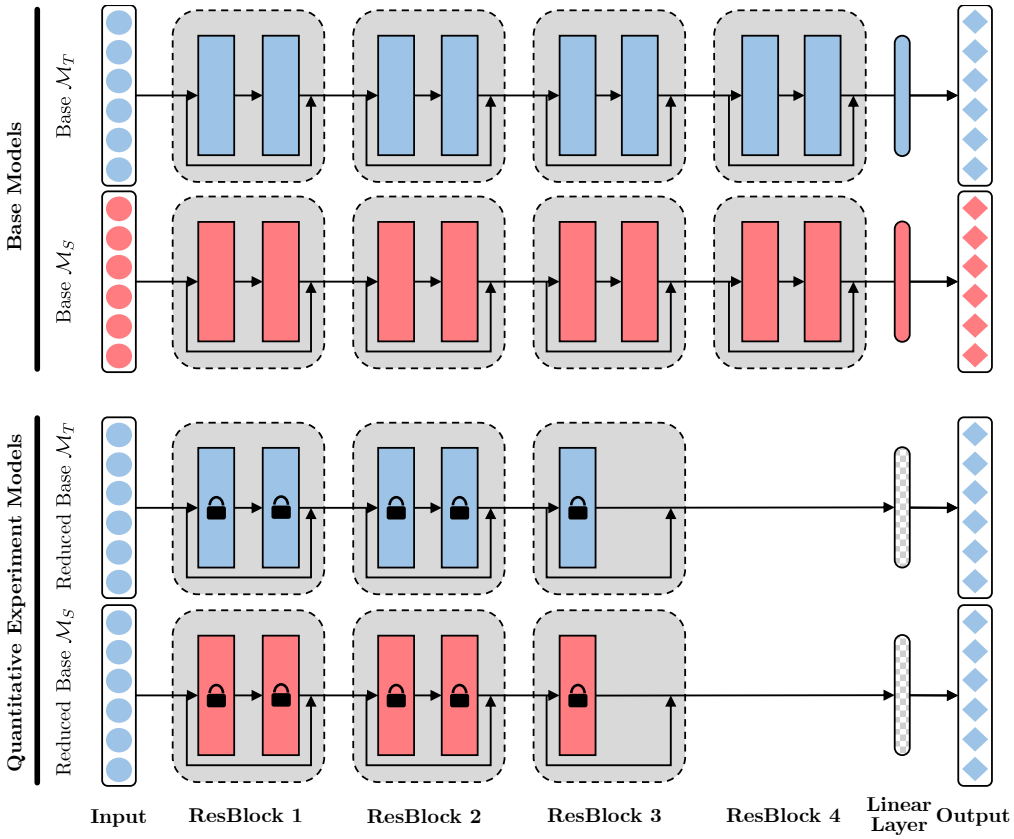


Figure 5.1: Visualization of the quantitative evaluation setup for evaluation level $l = 5$. Only l layers are kept in the reduced models and mapped directly to the dense layer. The dense layer is initialized with random weights. *Top two rows*: The base networks $\mathcal{M}_T$ and $\mathcal{M}_S$ are trained on the two synthetically generated distinct datasets from $\mathcal{D}_T$ and $\mathcal{D}_S$. All layers are initialized with random weights and then trained on their respective domain. *Third row:* The reduced version of base $\mathcal{M}_T$. The lower l layers from model $\mathcal{M}_T$ are frozen and mapped to the dense layer. The network is then trained on dataset $\mathcal{D}_T$. *Fourth row:* The reduced version of base $\mathcal{M}_S$. The lower l layers from model $\mathcal{M}_S$ are frozen and mapped to the dense layer. The network is then trained on dataset $\mathcal{D}_T$.

upper convolutional layers from $(l + 1)$ to L. The remaining convolutional layers are frozen and directly connected to the dense layer of the network. The dense layer of both models is initialized with random weights and is left unfrozen. This way, we create eight different models for both datasets that contain zero to seven of the convolutional layers from the respective original base model. We do not create an addition model with all eight convolutional layers from the base model because this would simply reproduce the complete base model. The intuition behind this approach is that we gradually increase the amount of knowledge that is retained from the base model. The performance gain resulting from an additionally retained layer then can be regarded as a measure for the amount of knowledge that this particular layer contains with regards to $\mathcal{D}_T$.

3. **Retraining of the reduced models on the target domain.** Both models $\mathcal{M}_T^*$ and $\mathcal{M}_S^*$ are retrained on the target dataset from domain $\mathcal{D}_T$ so that the dense layer can adapt to the frozen convolutional layers. The parameters for training are similar to the ones for the base models with the exception that we use a different learning rate schedule for the special case of $l = 0$ when we keep none of the convolutional layers and train the dense layer only. We found that the dense layer alone needs less training time to converge and tends to overfit with the previously used learning rate schedule. To avoid overfitting, the learning rate is therefore reduced earlier than for the other levels[1].

As for the previous experiments, we run this evaluation in both directions, i.e. once with B as source dataset and A as target dataset and once the other way round. For each model $\mathcal{M}_T^*$ and $\mathcal{M}_S^*$ we calculate the relative knowledge $\kappa(l)$ that is contained in every layer l of the model. We calculate the knowledge of one layer as the performance increase that is achieved by adding this layer to the network divided by the total performance difference between the complete base model and the dense layer only. This results in the following equation for the relative layer knowledge of some layer l:

$$\kappa(l) = \frac{\lambda(l-1) - \lambda(l)}{\lambda(0) - \lambda(L)}, l \in [1; L] \tag{5.1}$$

With the model loss $\lambda(l)$ as a function of the number l of kept convolutional layers. We use the same architecture as for the previous experiments, hence the total number of layers is $L = 8$. Note that the eighth loss value $\lambda(l = 8)$ is the loss of the original base model which represents the maximum performance that any of the generated reduced models can reach.

5.2 Experiment Results

The results for the quantitative experiment are depicted in figure 5.2 for the evaluation with dataset A as target dataset and in figure 5.3 for the evaluation with dataset B as target dataset. Upper plots show the loss values λ for all generated models over 10 repetitions. Bottom plots are the mean losses over all repetitions. The inset axes indicate the relative knowledge κ as per equation (5.1).
As expected, the loss generally drops with additional layers that we keep in the network. Based on the resulting figures, we conclude the following points:

- Independent of the dataset that the base model was initially trained on, the first two layers (i.e. the first residual block) show almost identical performance on any of the two datasets. This means that they generalize just as well on the dataset that they were originally trained on as on the respective other dataset. This is reflected

[1] Instead of dividing the learning rate by factor 10 after epochs 100 and 110 (cf. section 3.2.1), the learning rate is divided by the same factor after epochs 10 and 15 for the special case $l = 0$.

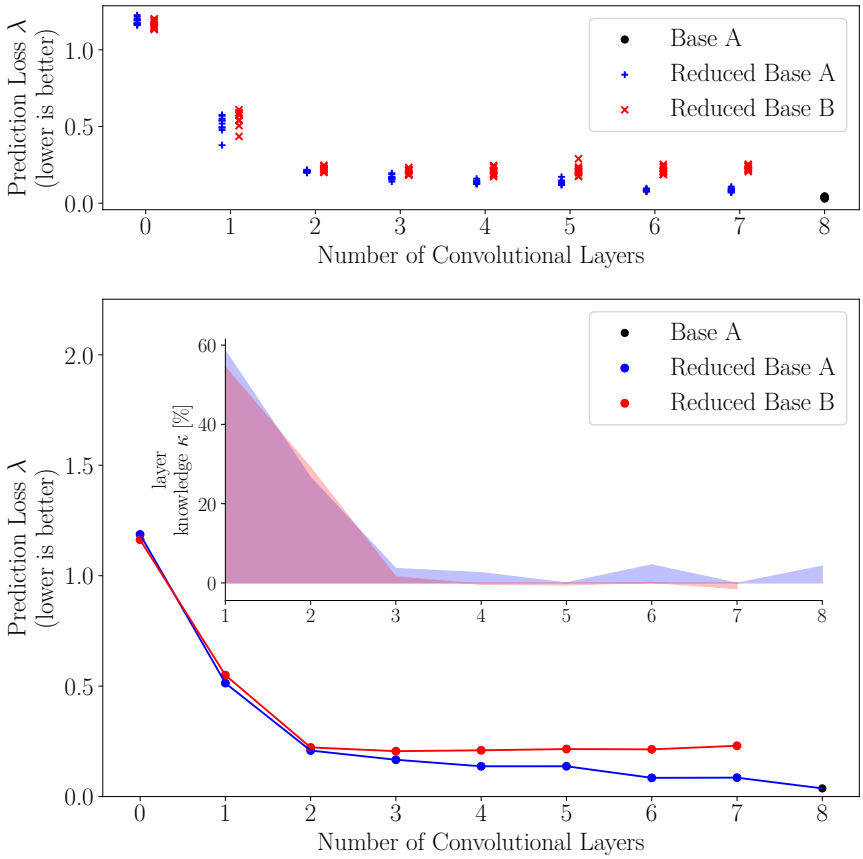


Figure 5.2: Loss plots of the quantitative experiment for the evaluation on dataset A as target dataset. Note that the plot for the reduced base B ends at layer 7 because the complete base B is not evaluated on dataset A.

in the similar loss curves of both reduced base models in figures 5.2 and 5.3. The divergence of the loss curves for layers 3 and higher indicates the dataset specificity of these layers which we observed already in the previous chapters.

- For both datasets, the first two layers (i.e. the first residual block) contain the most knowledge as per equation (5.1). This matches our results from the previous chapter where we found that the first residual block has the highest influence (cf. figure 4.3). We also find that layers three and deeper generally do not contribute much knowledge to predicting the dataset that they were not trained on (see red layer knowledge areas in figures 5.3 and 5.2). This explains why we observed a significant loss divergence during the qualitative experiments typically for transfer levels larger than 2. Moreover, this is the reason why the performance boost we observed in the Soft Feature Extraction Experiment from dataset B to A (see section 3.3.2) does not increase for $l > 2$ but stays practically constant. Layers 3 and deeper of a model trained on dataset B simply have not enough knowledge that benefits a target dataset A, as we can see in the layer knowledge plot for base B in figure 5.2.

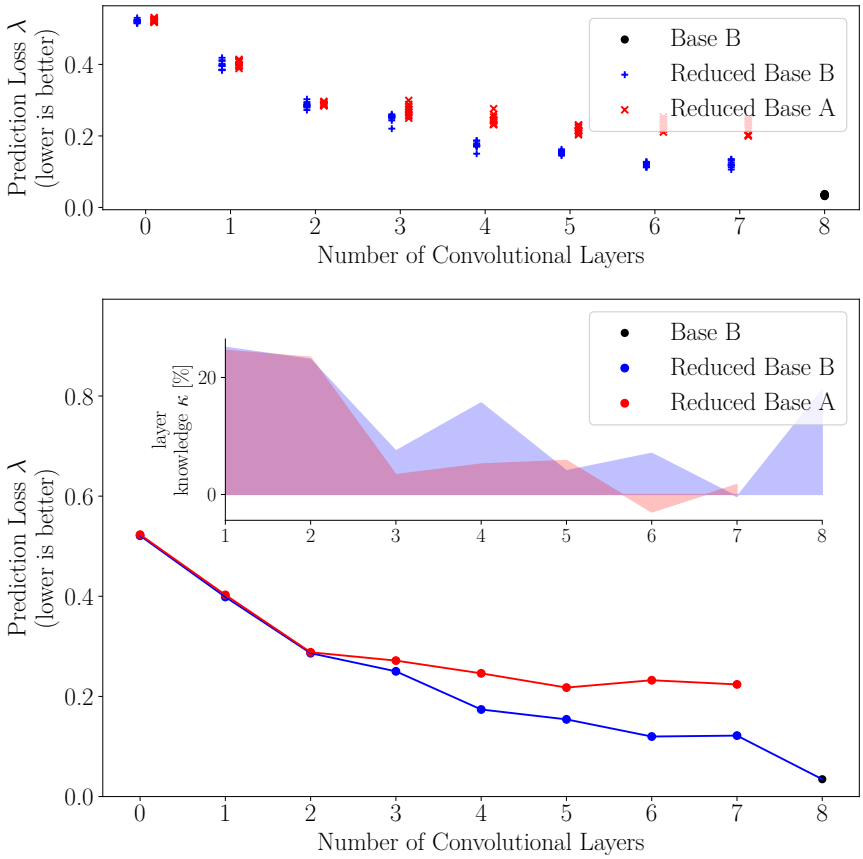


Figure 5.3: Loss plots of the quantitative experiment for the evaluation on dataset B as target dataset. Note that the plot for the reduced base A ends at layer 7 because the complete base A is not evaluated on dataset B.

- A large divergence in layer knowledge between the two reduced models can only be observed with B as target dataset (figure 5.3 for $l > 2$). This divergence is particularly noticeable in the second convolutional layer of residual blocks two to four, hence layers 4, 6 and 8 with a decreasing relative layer knowledge κ. This shows that the base model trained on dataset B has specific knowledge in deeper residual layers that is important for the prediction of this dataset, given the relatively large values of κ (e.g. around 15% for layer 4). This gives better insight into why a transfer from dataset A to dataset B did not yield as good performances as the other way round. The prediction performance for dataset B depends more on deep and specific layers than for dataset A. A model trained on dataset A does not contain this specific knowledge and therefore does not generalize well on dataset B. Our previous assumption that the different transfer performances result from different amounts of general information in both models therefore needs to be corrected. In fact, both models contain the same amount of general knowledge, as we can see from the κ values of layers 1 and 2. The first two layers of either base model perform equally

well on either target dataset. However, a prediction on dataset B also depends heavily on specific knowledge in deep layers that is not present in a model trained on dataset A. The reason for the different transfer performances therefore lies in the different amount of specific features required in deep layers.

- With A as target dataset, we also observe a divergence of κ for the two reduced models at the same layers 4, 6 and 8. However, the values for κ are small compared to the relative layer knowledge of the first two layers. A prediction of dataset A hence depends mostly on the first two layers which can be considered as general. This also explains the good transferability from dataset B to dataset A.

5.3 Discussion

With the results presented in this chapter, we have quantified how much knowledge the different layers in a TCN model learn. It became apparent that the first layers contribute most to the overall predictive power of the model, which is consistent with the activation analysis of the previous chapter. This is the case for both datasets that we investigated. However, we also observed that the knowledge contribution of the different layers depends highly on the dataset that is predicted. While one of the datasets requires almost exclusively knowledge from the first two general layers, prediction of the other dataset also needs specific knowledge in deep layers. Apart from the first residual block, the second convolutional layer in each block seems to be generally more important than the first layer. In the discussion of the qualitative experiment results in section 3.4, we formulated decision rules to help decide which kind of transfer should be selected under different circumstances. The main discriminator for these rules was whether the layer threshold is known at which the network transitions from general features to specific features. While a transfer can be more efficient and effective if this threshold is known, the difficulty lies in determining the layer at which the transition happens. This is particularly difficult because the threshold cannot be assumed to be general. For a model trained on some source dataset, the threshold depends on the relatedness of the source dataset and the chosen target dataset. It is therefore subject to change when a different target domain is chosen and does not depend on the source model alone. With the qualitative evaluation of layer transferability, we introduced an experiment that gives insight into where the threshold lies in the source model with regards to the target dataset. As we have seen, the divergence of the two loss curves of models $\mathcal{M}_S^*$ and $\mathcal{M}_T^*$ can also be regarded as an indicator for the threshold where layers transition to specific features. This information can benefit an effective and efficient transfer according to the decision rules that were formulated in section 3.4. However, more research is needed to evaluate whether this approach for identifying the threshold is also effective for small target datasets.

Chapter 6

Conclusion

In the present work, three aspects of Transfer Learning for time series using Temporal Convolutional Networks were explored. We qualitatively evaluated the transferability of layers in TCNs, visualized the knowledge that TCN layers contain and quantified the amount of knowledge that different layers contain. In this chapter, we conclude the work on these aspects by highglighting the most important findings and give an outlook on possible future work.

6.1 Results

The starting point for the evaluation of layer transferability for Temporal Convolutional Networks at IAV was the finding that the TCN architecture constitutes a promising approach to model engine test bench data for engine calibration. The shortcoming of TCNs so far was longer training times compared to the traditional approaches at IAV, namely Gaussian Process Models and parametric Volterra series, while not reducing the amount of necessary training data. For Convolutional Neural Networks, Transfer Learning approaches are used in practice to address both these points. The goal of this work was therefore to evaluate similar approaches for TCN architectures and to fill the research gap that existed for Transfer Learning in the area of time series modeling. The evaluation was guided by the following three research questions:

1. Can the knowledge learned in a TCN be transferred to other tasks and domains?
2. If parts of a TCN model are transferable, what kind of knowledge is encoded in the transferable features?
3. How much do the different layers in a TCN contribute to the overall predictive power of the model?

To answer the first research question, we performed a qualitative evaluation of layer transferability to find out whether layers in TCN models are transferable and in particular which parts of the network are suited to be transferred to a different task or domain. We found that a transfer between related domains is possible and even promises to benefit the performance of the target model under certain conditions. The actual performance on the target domain is influenced by three factors:

- The difference between the datasets may result in a performance boost or decrease of a transfer model compared to a model that was directly trained on the target dataset. The amount of general information in a dataset and the degree of specificity of the information in a dataset can result in a performance boost or decrease.
- Deep layers were shown to be more specific to the dataset they were trained on and therefore are not well transferable.
- We found that even if layers are specific, the knowledge they contain is still to a certain extent relevant for a related target dataset. Such layers have proven to perform better when transferred than randomly initialized weights.

Apart from these three findings, we also evaluated the duration it took to train the transfer models. A significant improvement could not be observed with a concurrent satisfactory transfer model performance. However, we were able to show that a weight transfer has no negative effect on the training duration if general layers are transferred. Our results therefore do not suggest a benefit in terms of training time. Still, they encourage a transfer of weights since this may result in a performance boost and reduces the number of parameters to train.

For the second research question, we performed a visualization of the knowledge that TCN layers learn. For this purpose, we visualized the intermediate activations of layers. The resulting series suggested different tasks that neurons perform within the layers. Such tasks included smoothing, inverting, shifting and filtering. Because these visual interpretations are not easily done for all intermediate activations, we also quantified the importance of different neurons and layers within the network. We found that the first layers of the network have the strongest impact on the model output. These are also the layers that we found to be general which explains the successful weight transfers. An evaluation of the impact that different channels - hence, neurons - have on the final output of the model showed that there are some neurons with relatively large impact and the activation distribution has a tendency towards a heavy tail.

Finally, to answer the third research question, we conducted a quantitative evaluation of layer transferability. We quantified the amount of knowledge that layers in a TCN contain by measuring the loss reduction on the prediction of the target dataset that a single network layer accounts for. This evaluation substantiated the findings to the previous questions that the first, general layers contribute most to the overall performance of the models. At the same time, the degree to which also specific, deeper layers have a significant knowledge contribution depends on the individual dataset to be predicted. Moreover, the presented approach provides a useful and efficient tool to find the threshold within the network where layers transition from general to specific.

6.2 Future Work

Based on the results of our research, we highlight possible future aspects that we find worth investigating in the future to further improve our knowledge of the mechanisms behind transferring knowledge among TCN models.
Based on the results of our qualitative transferability evaluation, we suggested to transfer entire networks and to freeze the low, general layers up to the threshold where the network transitions from general to specific. Besides shorter training times, this suggestion was based on the intuition that freezing layers leads to less trainable parameters and reduces the risk of overfitting the dataset. This assumption should be verified in the future by comparing differently sized networks and their tendency to overfit a given dataset.
The given transfer suggestions differ based on whether the transition threshold is known or not. With our quantitative evaluation of knowledge in TCN layers, we also introduced an approach to efficiently find this transition threshold given two datasets. However, unlike in our experiments, the target dataset is typically small compared to the source dataset. An evaluation of this experiment for small target datasets should therefore further substantiate our findings.
Another important next step is a transferability evaluation on real-world datasets. While the use of synthetic data allowed us to steer the complexity and relatedness of datasets, the practical use of transferring TCN weights should be confirmed with real-world data. A next step to evaluate layer transferability of TCNs at IAV therefore is to use engine test bench data to generate engine models and to evaluate transfers among these models. An interesting aspect will be the performance of transfers between models based on engines that are physically similar or less similar to find which layers and features can be regarded as general and specific in the area of time series modeling for engine calibration.
Models of combustion engines are also typically generated from multivariate input series, with the number of parameters used for calibration ranging usually between 12 and 15. In the presented experiments, only the univariate case was considered. Therefore, future work should also evaluate transferability between models with multivariate inputs. It will be particularly interesting to investigate the success of weight transfers between models with different features, known as heterogeneous Transfer Learning [8]. If such transfers perform well due to common general features in the hidden layers, not only Transfer Learning for TCNs may become more efficient but also weight sharing between channels may be an option, resulting in networks with less parameters.
With regards to layer visualization, we gained some first insights to better understand the features that TCNs learn. The presented results suggest that neurons learn tasks like smoothing, inverting, filtering and others. However, a solid mathematical background beyond a visual interpretation is desirable. As a promising next step, the learned weights should be analyzed in more detail. For example, analyzing the filter weights using bode plots could be a good approach to explain the function of neurons from a signal processing perspective. More elaborate visualization techniques than the activation visualization may also benefit the understanding of the knowledge encoded in TCN layers. Activation

maximization [11] and deconvolutional networks [25] are promising approaches in this regard.
We found that some datasets are more suitable for generating a source model than others. This raises the question whether a dataset can be found or generated to pre-train a model that can serve as a general source model for knowledge transfers in the area of time series modeling. For the domain of Natural Language Processing (NLP), Howard and Ruder [15] suggest to pre-train a language model on a very general dataset to be used as a universal source model for NLP. Such a universal model would generalize well on many different downstream tasks and domains. A similar model is also desirable for sequence modeling and finding suitable datasets and tasks to generate this model should be addressed by future research. In this regard, a systematic quantitative evaluation of the transfer performance between common tasks (e.g. regression and classification) or domains may be helpful.

We hope that with the research presented in this thesis, we were able to establish a useful start for research in the the domain of Transfer Learning for time series modeling with Temporal Convolutional Networks. We are confident that our findings will benefit the efficient generation of time series models in the future and that they will show practical benefits where model generation was difficult or impossible up until now due to a lack of training data. At the same time, given the aforementioned many remaining research issues worth investigating, we encourage further research in the area of Transfer Learning for time series modeling.

Bibliography

[1] Ratnadip Adhikari and Ramesh K. Agrawal. "An introductory study on time series modeling and forecasting". In: *arXiv preprint arXiv:1302.6613* (2013). URL: https://arxiv.org/abs/1302.6613.

[2] Ronald L. Allen and Duncan Mills. *Signal Analysis: Time, Frequency, Scale, and Structure*. 1st ed. Piscataway: Wiley-IEEE Press, Jan. 2004. DOI: 10.1002/047166037X.ch1.

[3] Shaojie Bai, J. Zico Kolter, and Vladlen Koltun. "An Empirical Evaluation of Generic Convolutional and Recurrent Networks for Sequence Modeling". In: *arXiv preprint arXiv:1803.01271* (2018). URL: https://arxiv.org/abs/1803.01271.

[4] Peter Bloomfield. *Fourier Analysis of Time Series: An Introduction*. 2nd ed. New York: John Wiley & Sons, 2000.

[5] Peter J. Brockwell and Richard A. Davis. *Introduction to Time Series and Forecasting*. 2nd ed. Cham: Springer International Publishing, 2002. ISBN: 978-3-319-29854-2. DOI: 10.1007/978-3-319-29854-2_3. URL: https://doi.org/10.1007/978-3-319-29854-2_3.

[6] Marco Castelluccio et al. "Land use classification in remote sensing images by convolutional neural networks". In: *arXiv preprint arXiv:1508.00092* (2015). URL: https://arxiv.org/abs/1508.00092.

[7] Kyunghyun Cho et al. "Learning phrase representations using RNN encoder-decoder for statistical machine translation". In: *arXiv preprint arXiv:1406.1078* (2014). URL: https://arxiv.org/abs/1406.1078.

[8] Oscar Day and Taghi M. Khoshgoftaar. "A survey on heterogeneous transfer learning". In: *Journal of Big Data* 4.1 (Sept. 2017), p. 29. ISSN: 2196-1115. DOI: 10.1186/s40537-017-0089-0. URL: https://doi.org/10.1186/s40537-017-0089-0.

[9] Jia Deng et al. "Imagenet: A large-scale hierarchical image database". In: *2009 IEEE conference on computer vision and pattern recognition*. IEEE. 2009, pp. 248–255.

[10] Jay L. Devore and Kenneth N. Berk. *Modern Mathematical Statistics with Applications*. New York: Springer New York, 2012. ISBN: 978-1-4614-0391-3. DOI: 10.1007/978-1-4614-0391-3_5. URL: https://doi.org/10.1007/978-1-4614-0391-3_5.

[11] Dumitru Erhan et al. "Visualizing Higher-Layer Features of a Deep Network". In: *University of Montreal* (2009).

[12] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. http://www.deeplearningbook.org. MIT Press, 2016.

[13] Alex Graves. *Supervised Sequence Labelling with Recurrent Neural Networks*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012. ISBN: 978-3-642-24797-2. DOI: 10.1007/978-3-642-24797-2_4. URL: https://doi.org/10.1007/978-3-642-24797-2_4.

[14] Sepp Hochreiter and Jürgen Schmidhuber. "Long short-term memory". In: *Neural computation* 9.8 (1997), pp. 1735–1780.

[15] Jeremy Howard and Sebastian Ruder. "Universal language model fine-tuning for text classification". In: *arXiv preprint arXiv:1801.06146* (2018). URL: https://arxiv.org/abs/1801.06146.

[16] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. "ImageNet Classification with Deep Convolutional Neural Networks". In: *Advances in Neural Information Processing Systems 25*. Ed. by F. Pereira et al. Curran Associates, 2012, pp. 1097–1105. URL: http://papers.nips.cc/paper/4824-imagenet-classification-with-deep-convolutional-neural-networks.pdf.

[17] Colin Lea et al. "Temporal convolutional networks for action segmentation and detection". In: *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*. 2017, pp. 156–165.

[18] Jonathan Long, Evan Shelhamer, and Trevor Darrell. "Fully Convolutional Networks for Semantic Segmentation". In: *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2015, pp. 3431–3440.

[19] Christopher Olah. *Understanding LSTM Networks*. 2015. URL: http://colah.github.io/posts/2015-08-Understanding-LSTMs/ (visited on 05/07/2019).

[20] Aaron van den Oord et al. "WaveNet: A Generative Model for Raw Audio". In: *arXiv preprint arXiv:1609.03499* (2016). URL: https://arxiv.org/abs/1609.03499.

[21] Sinno Jialin Pan and Qiang Yang. "A survey on transfer learning". In: *IEEE Transactions on knowledge and data engineering* 22.10 (2009), pp. 1345–1359.

[22] Steven W. Smith. *The scientist and engineer's guide to digital signal processing*. San Diego: California Technical Publishing, 1997.

[23] Alexander Waibel, Tosi-Iiyuki Hanazawa, and Geoffrey I-Iinton. "Phoneme Recognition Using Time-Delay Neural Networks". In: *IEEE Transactions on Acoustics, Speech, and Signal Processing* 37.3 (1989).

[24] Jason Yosinski et al. "How transferable are features in deep neural networks?" In: *Advances in neural information processing systems*. 2014, pp. 3320–3328.

[25] Matthew D. Zeiler and Rob Fergus. "Visualizing and Understanding Convolutional Networks". In: *Computer Vision – ECCV 2014*. Ed. by David Fleet et al. Cham: Springer International Publishing, 2014, pp. 818–833. ISBN: 978-3-319-10590-1.

[26] Matthew D. Zeiler, Graham W. Taylor, and Rob Fergus. “Adaptive deconvolutional networks for mid and high level feature learning”. In: *2011 International Conference on Computer Vision*. Nov. 2011, pp. 2018–2025. DOI: 10.1109/ICCV.2011.6126474.

Appendix A

Supplementary Material: Qualitative Evaluation of Layer Transferabilty

In this appendix chapter, we provide additional material for the qualitative evaluation of layer transferability from chapter 3. In section A.1, we provide loss plots for the three experiments Hard Feature Extraction, Soft Feature Extraction and Full Weight Initialization with partial Freeze in addition to those presented in chapter 3. Section A.2 contains probability plots that were generated to check for normality of the MSE sample distributions.

A.1 Loss Plots

In this section, we provide loss plots for the three experiments Hard Feature Extraction, Soft Feature Extraction and Full Weight Initialization with partial Freeze in addition to those presented in chapter 3. Section A.1.1 contains loss plots for the transfer from dataset B to dataset A with reduced datasets. Section A.1.2 contains loss plots for the transfer from dataset A to dataset B with complete datasets. In section A.1.3, we present loss plots for the transfer from dataset A to dataset B with reduced datasets.

A.1.1 Transfer from Dataset B to Dataset A - Reduced Datasets

Figures A.1 and A.2 contain the results for the Soft Feature Extraction and the Full Weight Initialization with partial Freeze with reduced datasets A and B, using A as target dataset and B as source dataset.

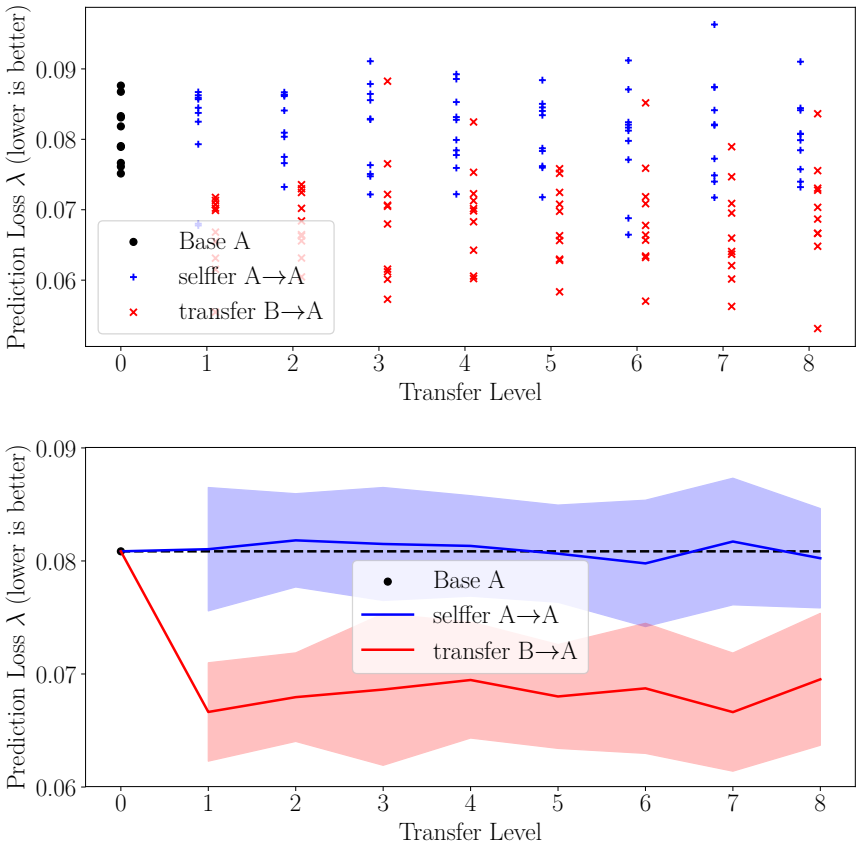


Figure A.1: Loss plot for the Soft Feature Extraction experiment from dataset B to dataset A with reduced datasets.

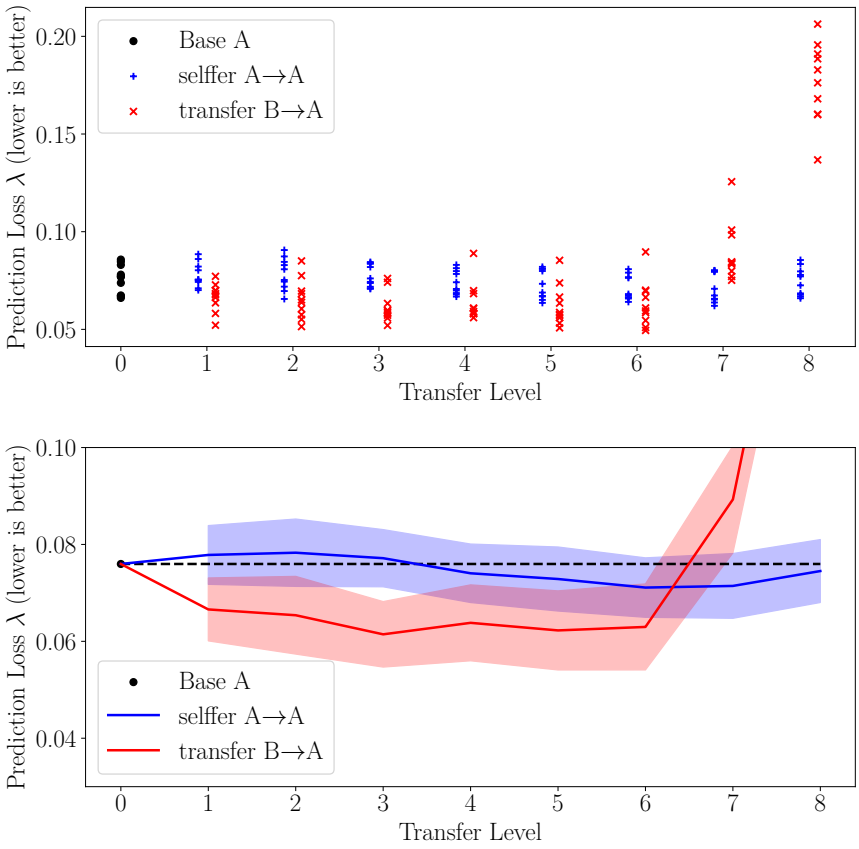


Figure A.2: Loss plot for the Full Weight Initialization with partial Freeze experiment from dataset B to dataset A with reduced datasets.

A.1.2 Transfer from Dataset A to Dataset B - Complete Datasets

Figures A.3 and A.4 contain the results for the Hard Feature Extraction and the Full Weight Initialization with partial Freeze with complete datasets A and B, using B as target dataset and A as source dataset.

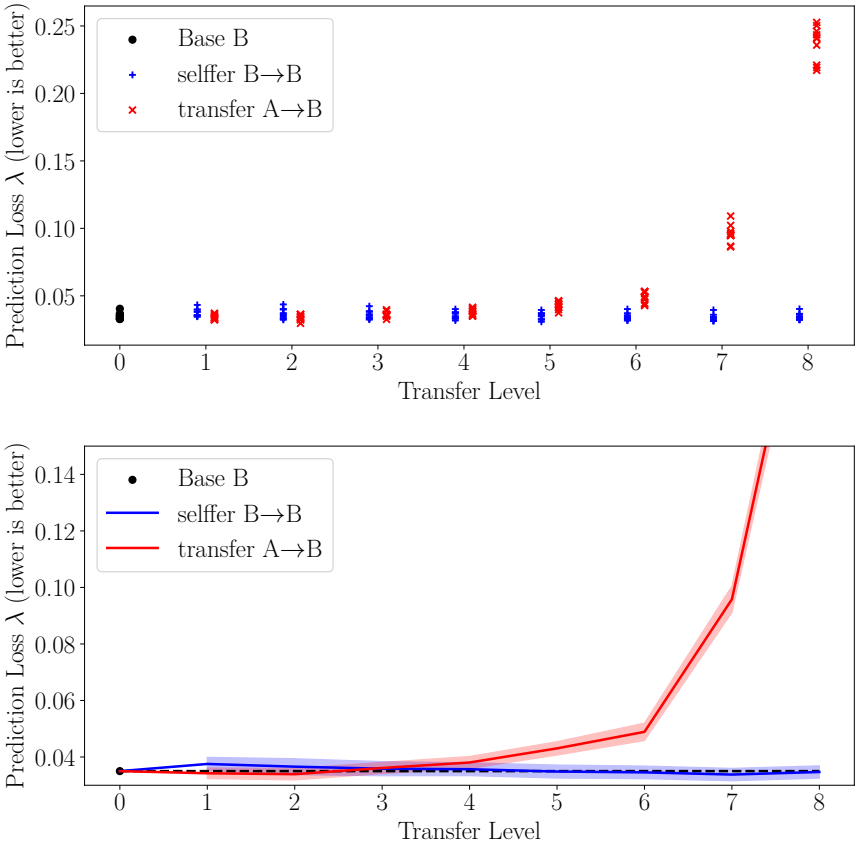


Figure A.3: Loss plot for the Hard Feature Extraction experiment from dataset A to dataset B with complete datasets.

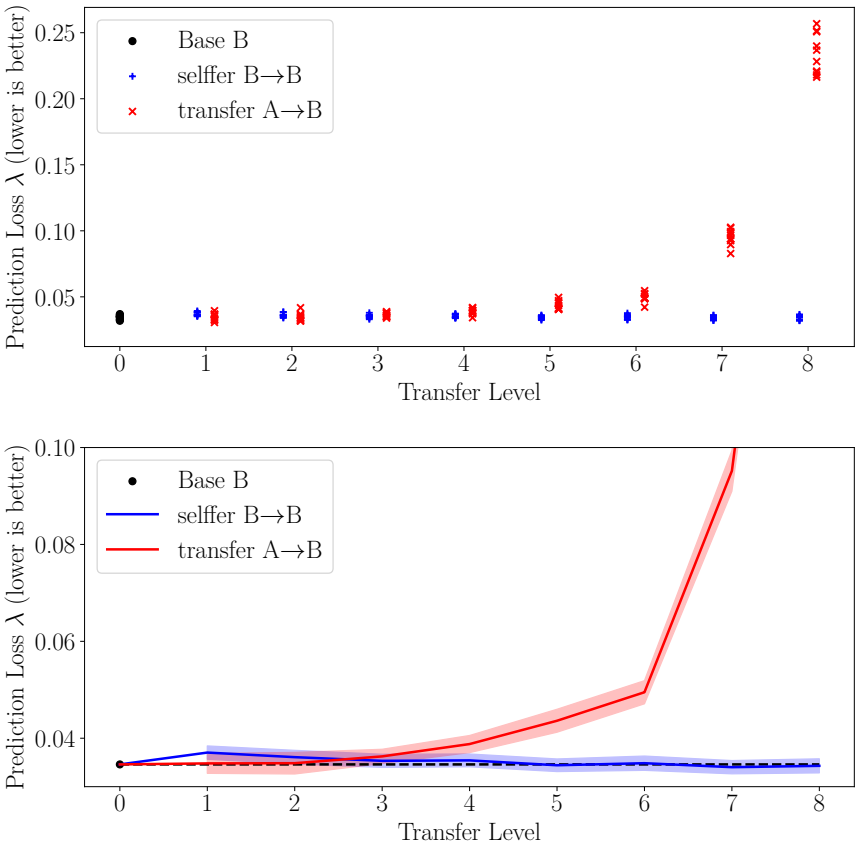


Figure A.4: Loss plot for the Full Weight Initialization with partial Freeze experiment from dataset A to dataset B with complete datasets.

A.1.3 Transfer from Dataset A to Dataset B - Reduced Datasets

Figures A.5, A.7 and A.6 contain the results for the Hard Feature Extraction, the Soft Feature Extraction and the Full Weight Initialization with partial Freeze experiments with reduced datasets A and B, using A as target dataset and B as source dataset.

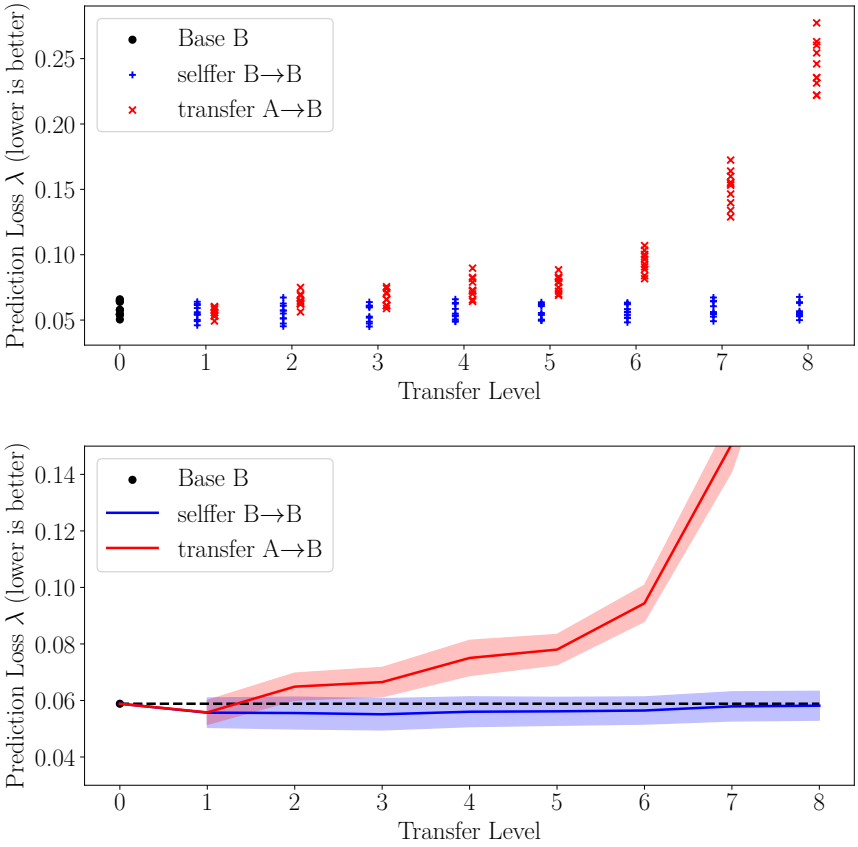


Figure A.5: Loss plot for the Hard Feature Extraction experiment from dataset A to dataset B with reduced datasets.

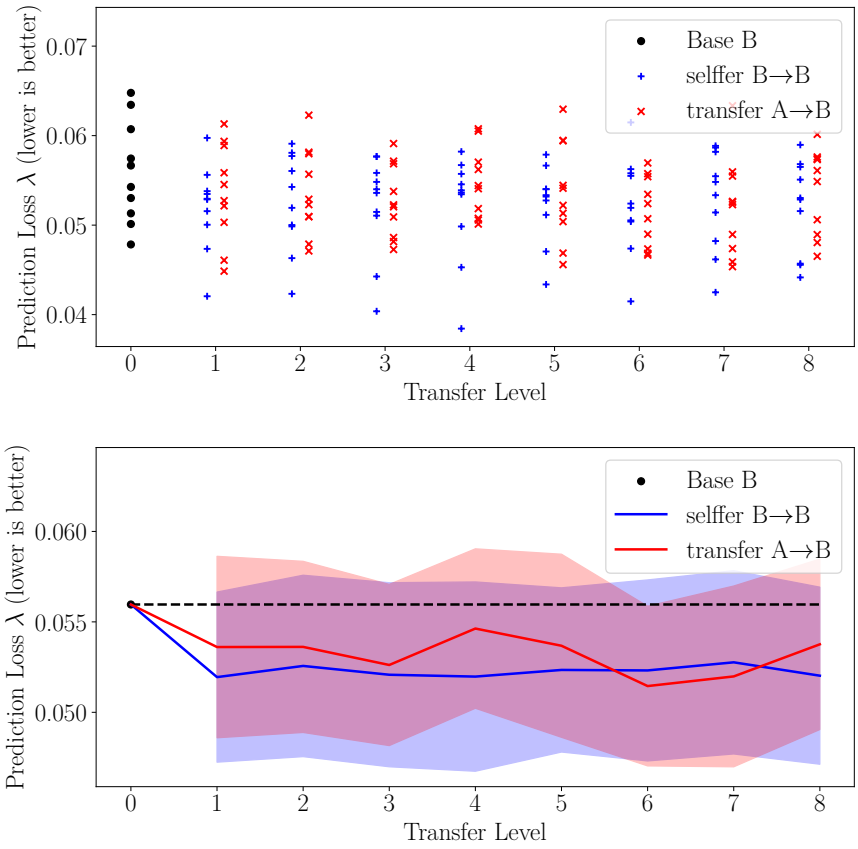


Figure A.6: Loss plot for the Soft Feature Extraction experiment from dataset A to dataset B with reduced datasets.

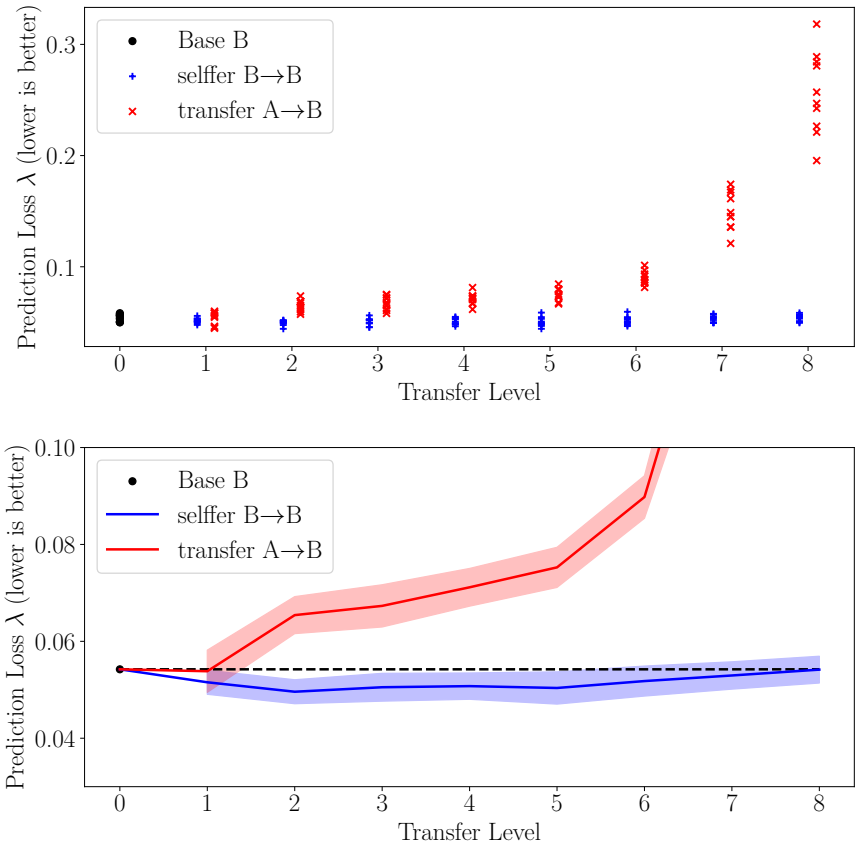


Figure A.7: Loss plot for the Full Weight Initialization with partial Freeze experiment from dataset A to dataset B with reduced datasets.

A.2 Probability Plots

In this appendix section, we provide probability plots that can be used to check for normality of the MSE distributions from all base, selffer and transfer models that are generated throughout the qualitative transferability experiments. Probability plots are only shown for models trained on complete datasets A and B. All experiments are repeated ten times, resulting in a sample size of ten for each MSE sample. Linearity of a probability plot suggests normality of the respective distribution. Note that Devore and Berk [10] highlight that samples of size less than 30 can show a nonlinear pattern even if they are drawn from a normal distribution and suggest to only consider very strong deviation from linearity in the probability plot as evidence against normality. We therefore assume linearity for all samples shown below. Section A.2.1 contains probability plots for all experiments with B as source dataset and A as target dataset. Probability plots for all experiments with B as source dataset and A as target dataset are presented in section A.2.2

A.2.1 Transfer from Dataset B to Dataset A

In this section, we provide probability plots for the average MSE losses of the qualitative transferability experiments conducted with B as source dataset and A as target dataset. Figures A.8 through A.11 contain the probability plots for the Hard Feature Extraction, the Soft Feature Extraction, the Full Weight Initialization with partial Freeze and the Random Reference experiments.

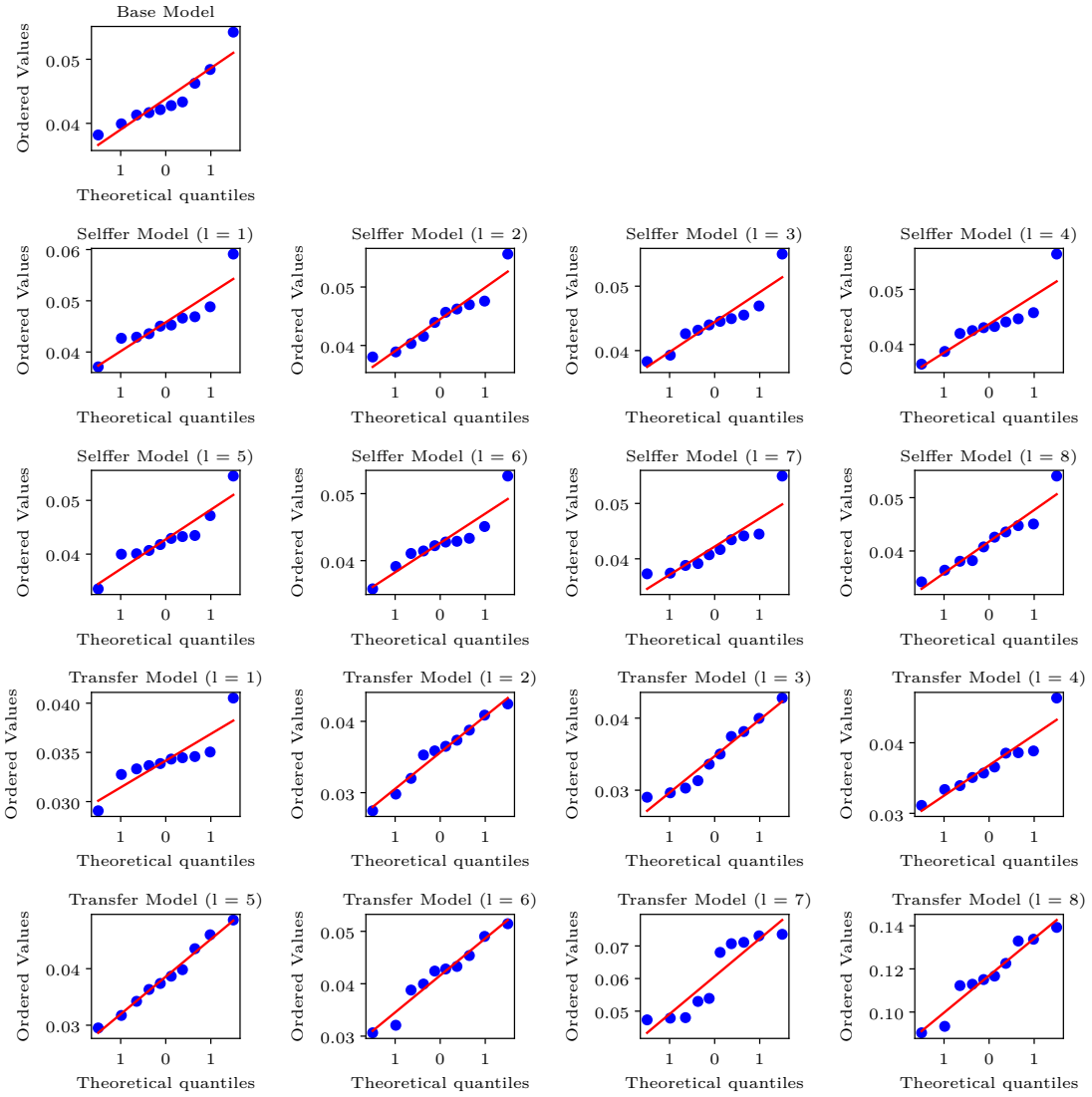


Figure A.8: Probability plot for the Hard Feature Extraction experiment from dataset B to dataset A with complete datasets.

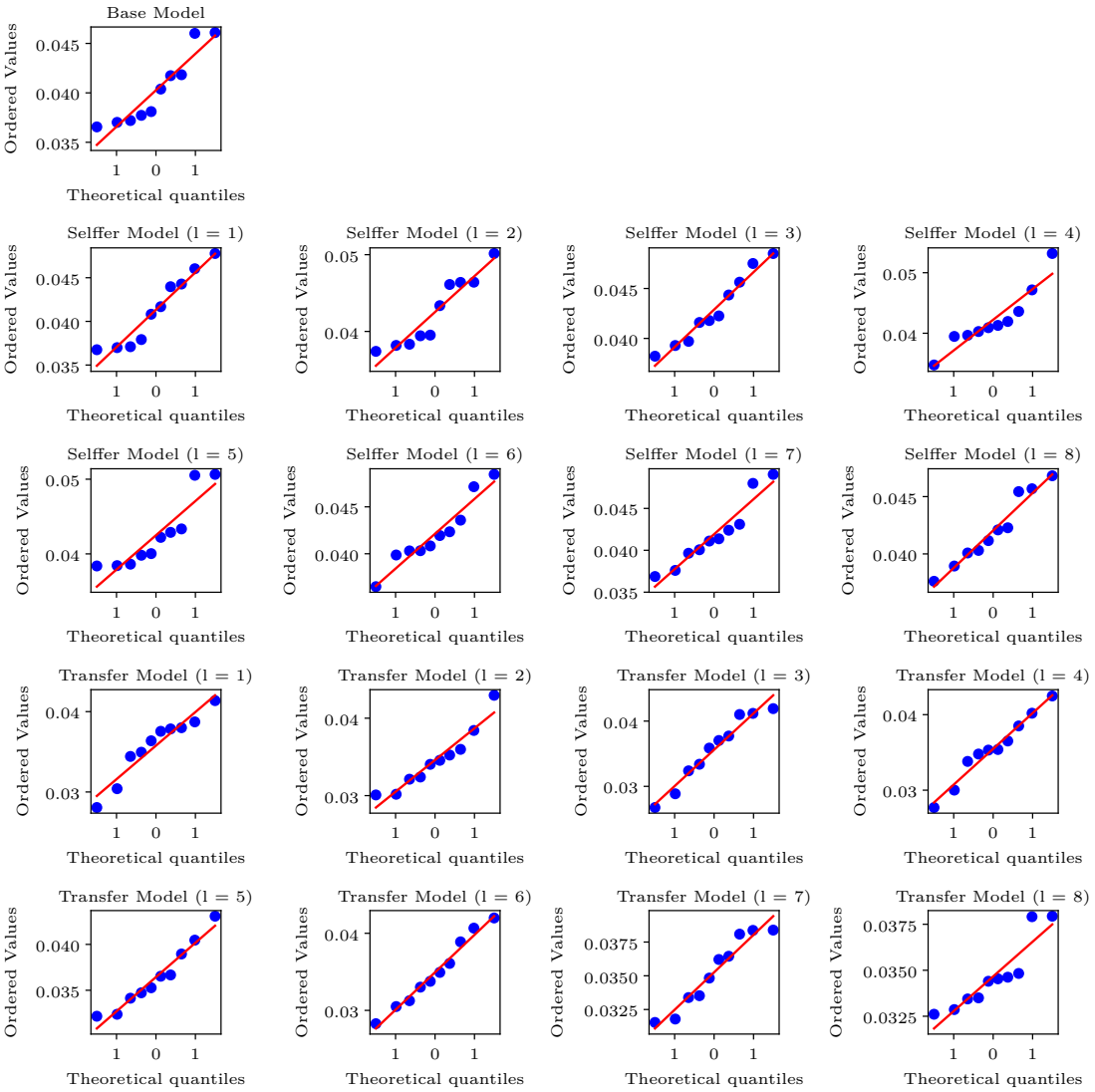


Figure A.9: Probability plot for the Soft Feature Extraction experiment from dataset B to dataset A with complete datasets.

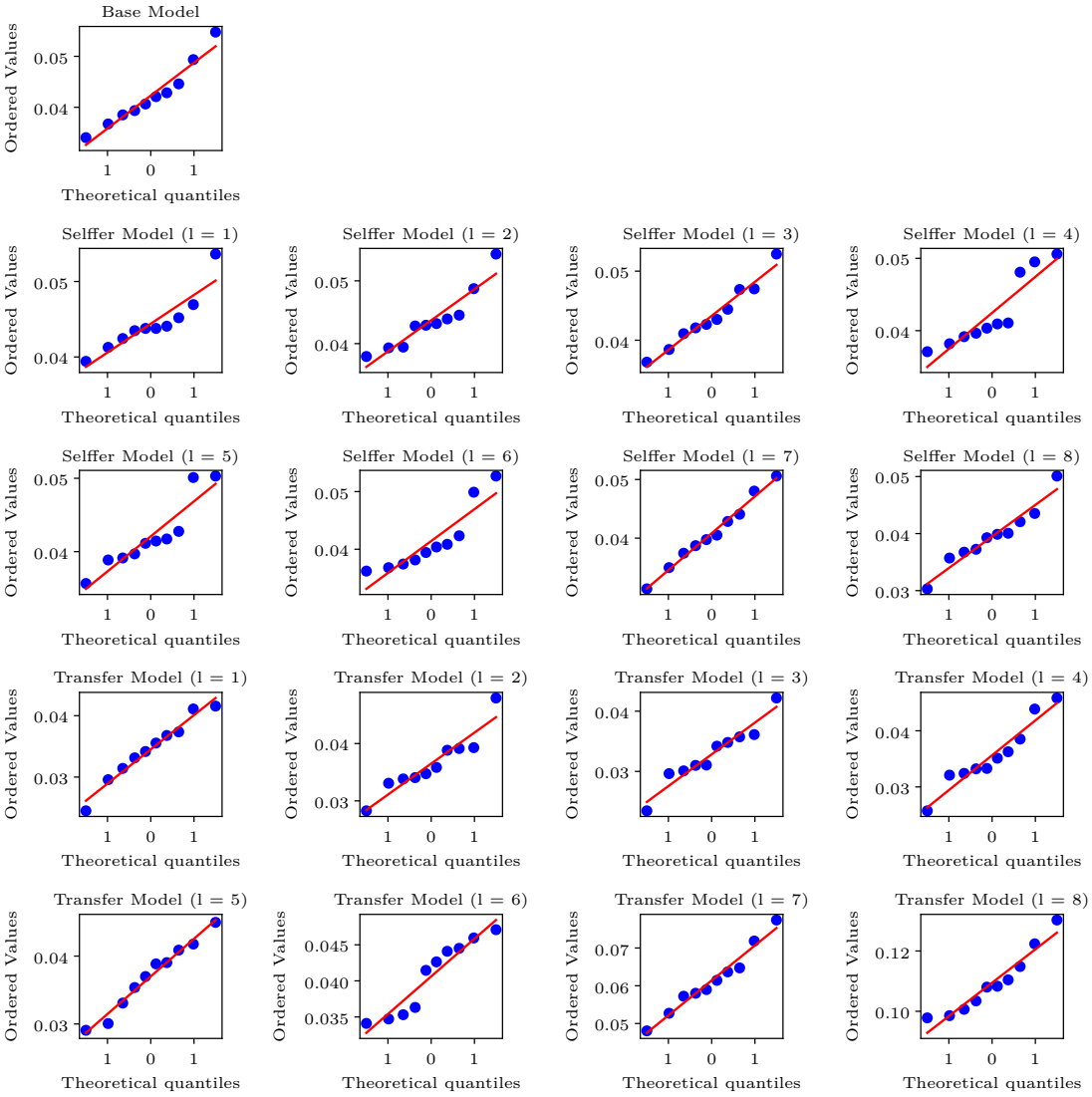


Figure A.10: Probability plot for the Full Weight Initialization with partial Freeze experiment from dataset B to dataset A with complete datasets.

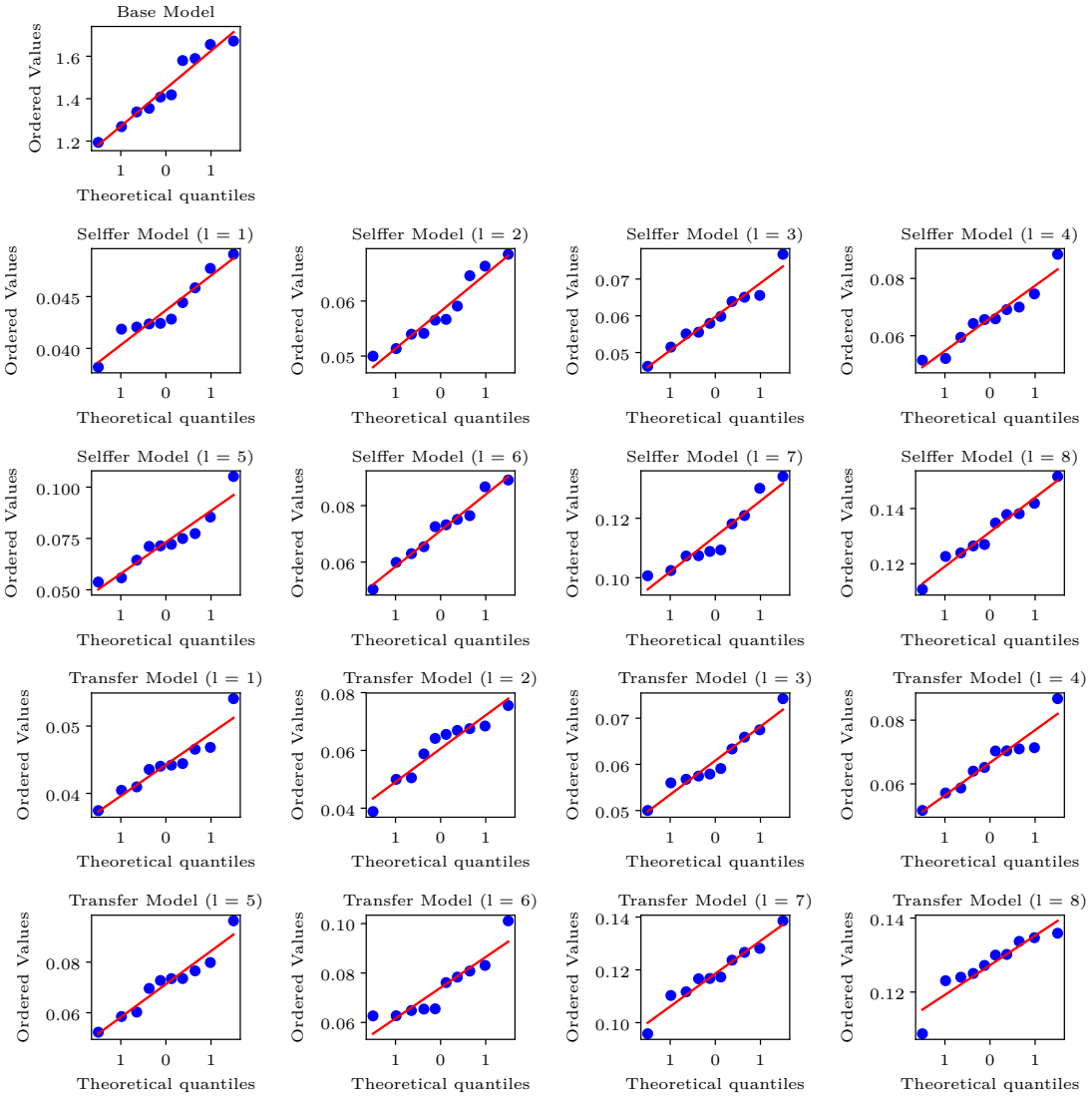


Figure A.11: Probability plot for the random reference experiment from random weights to dataset A with complete datasets.

A.2.2 Transfer from Dataset A to Dataset B

In this section, we provide probability plots for the average MSE losses of the qualitative transferability experiments conducted with A as source dataset and B as target dataset. Figures A.12 through A.15 contain the probability plots for the Hard Feature Extraction, the Soft Feature Extraction, the Full Weight Initialization with partial Freeze and the Random Reference experiments.

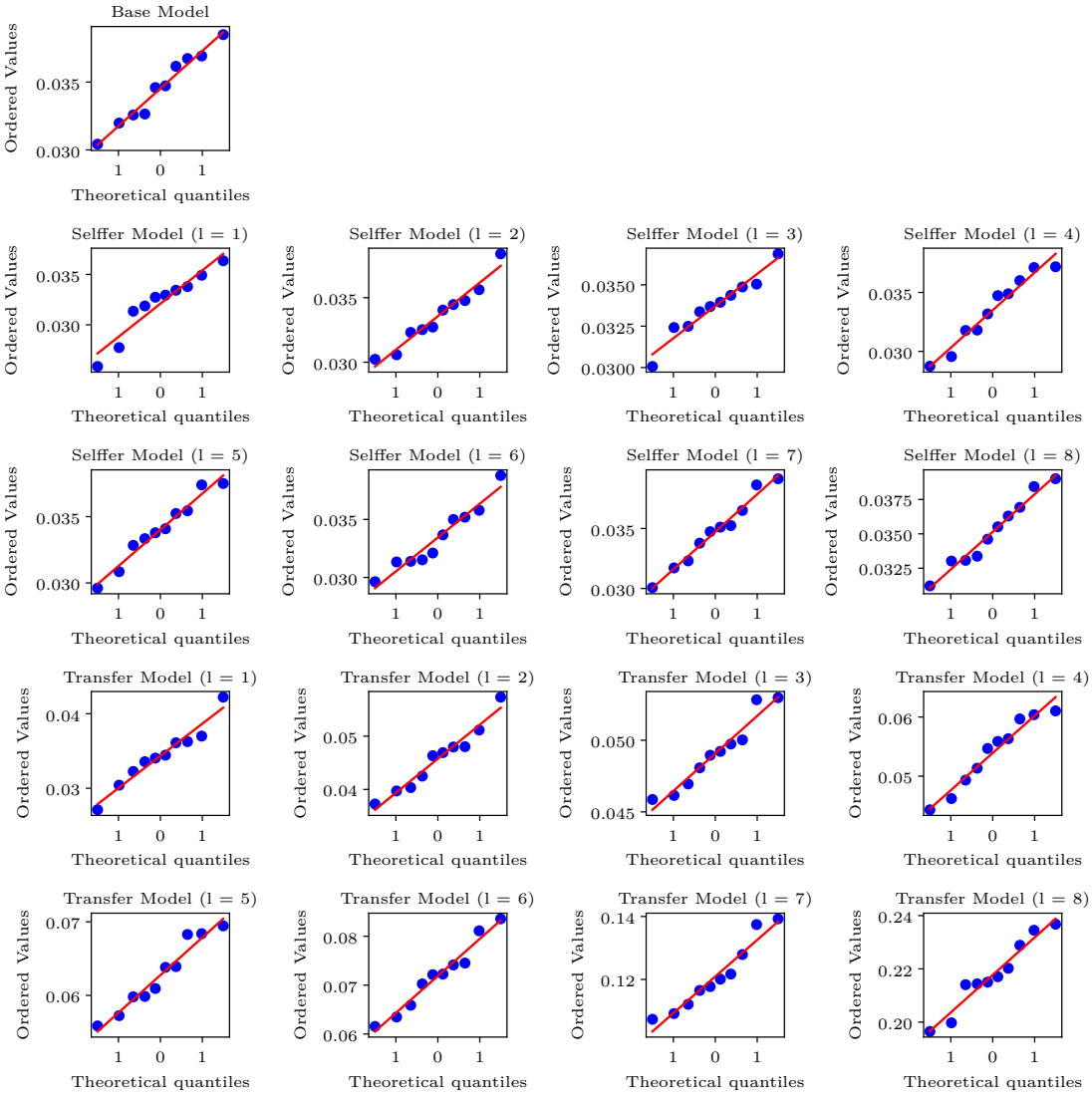


Figure A.12: Probability plot for the Hard Feature Extraction experiment from dataset A to dataset B with complete datasets.

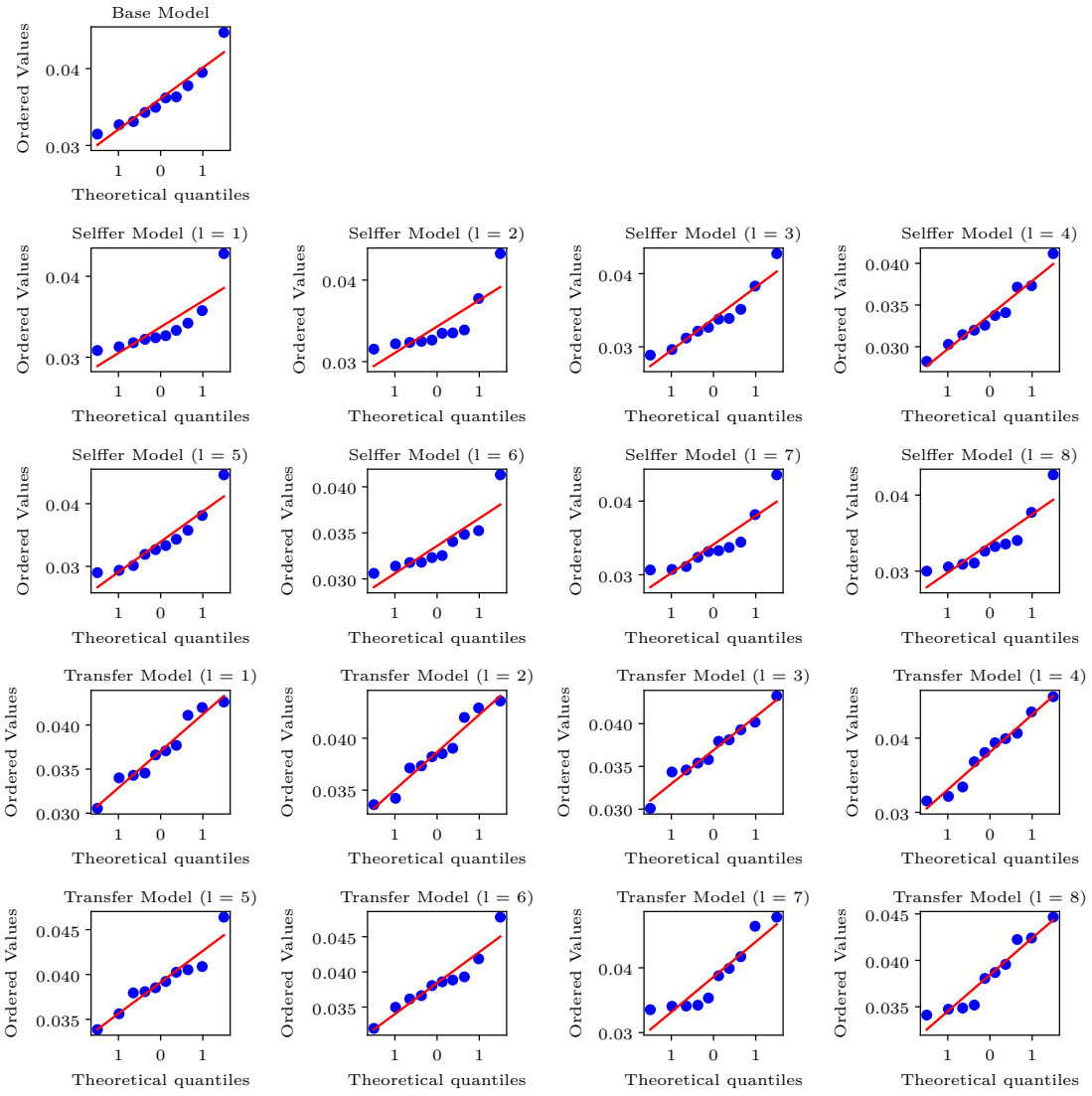


Figure A.13: Probability plot for the Soft Feature Extraction experiment from dataset A to dataset B with complete datasets.

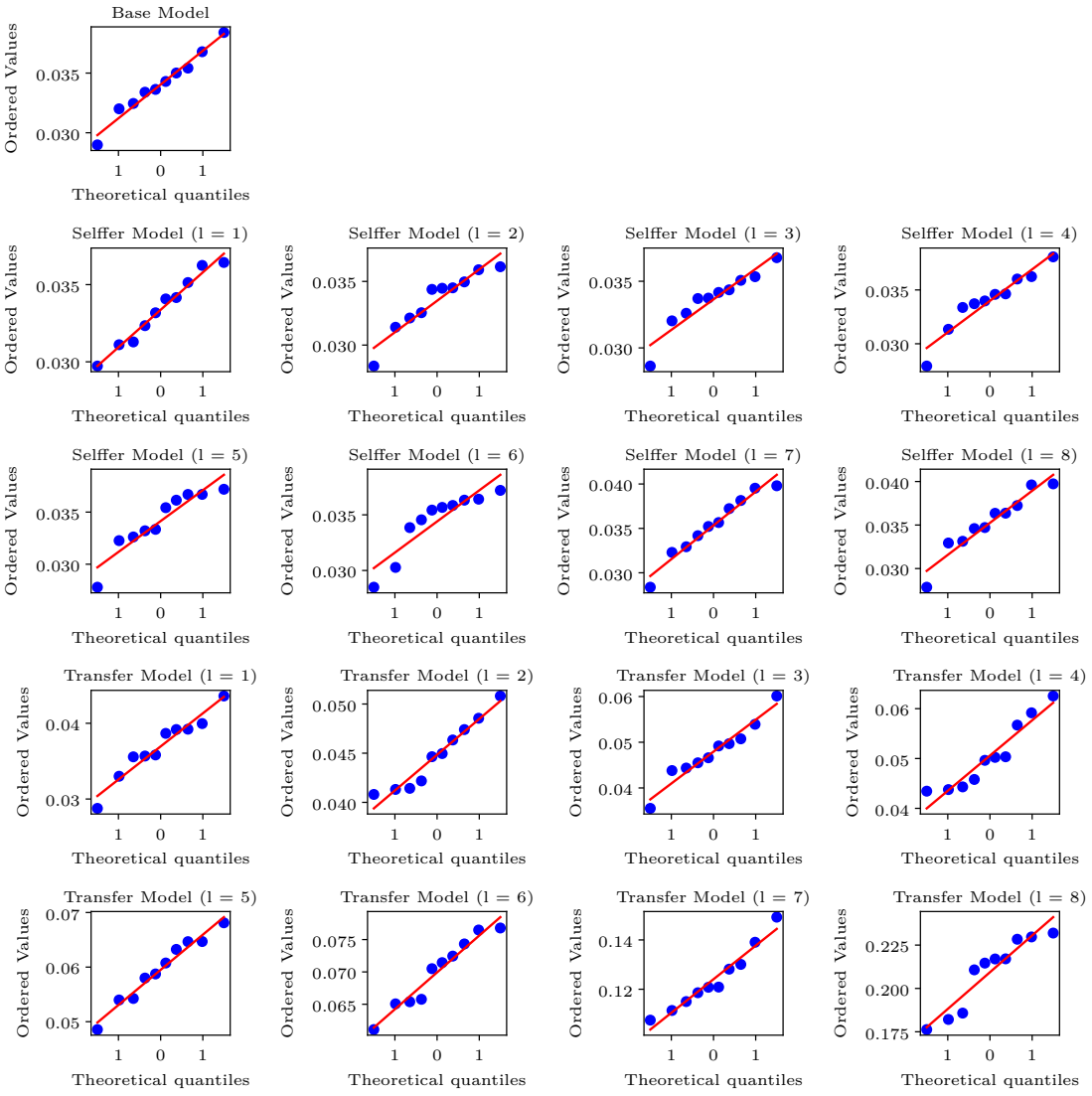


Figure A.14: Probability plot for the Full Weight Initialization with partial Freeze experiment from dataset A to dataset B with complete datasets.

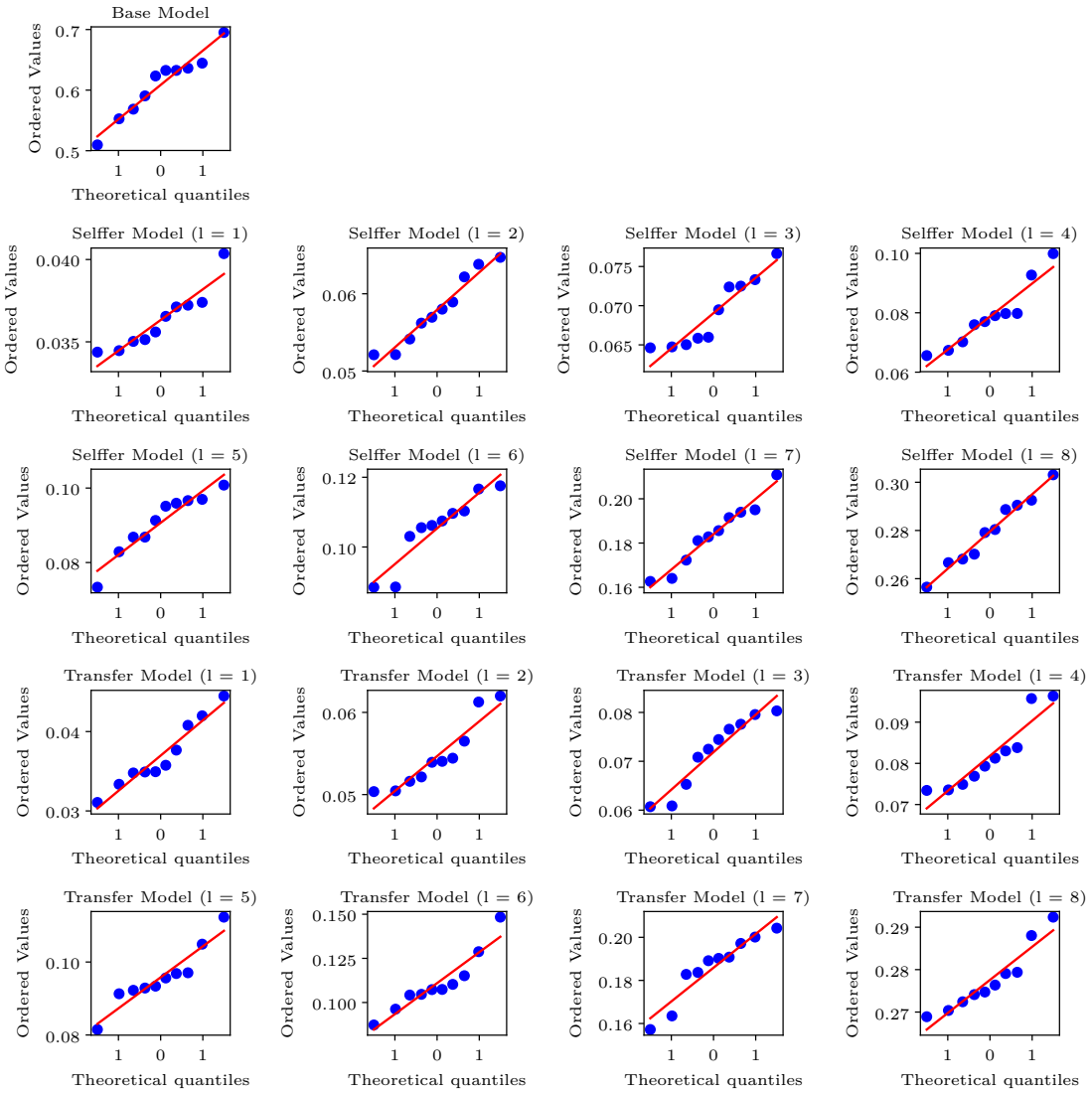


Figure A.15: Probability plot for the random reference experiment from random weights to dataset B with complete datasets.

Appendix B

Supplementary Material: Knowledge Visualization

In this appendix chapter, we provide additional material for the knowledge visualization from chapter 4. Section B.1 contains activation plots for models trained on datasets A and B for all fifty channels and eight different input sequences. In section B.2, we provide an influence visualization as well as an influence table for the horizontal evaluation on dataset A. Finally, section B.3 contains the vertical channel evaluation for dataset A.

B.1 Activation Plots

This appendix section contains activation plots for models trained on datasets A and B for all fifty channels and eight different input sequences. Activation plots of a model trained on dataset A are presented in section B.1.1 and those for a model trained on dataset B in section B.1.2. For both datasets, we provide activation plots for the following input signals: constant, dirac function, linear function, sawtooth, sine, square wave and a random test sequence from the dataset. Note that input signals have a length of 200 time steps, resulting in activations of the same length. However, for better readability, only the first 50 time steps of the neuron activations are plotted.

B.1.1 Dataset A

The following eight activation plots were obtained by exciting a TCN model trained on dataset A with the respective input sequences.

Figure B.1: Modification activation plots of a model trained on dataset A to a constant input signal.

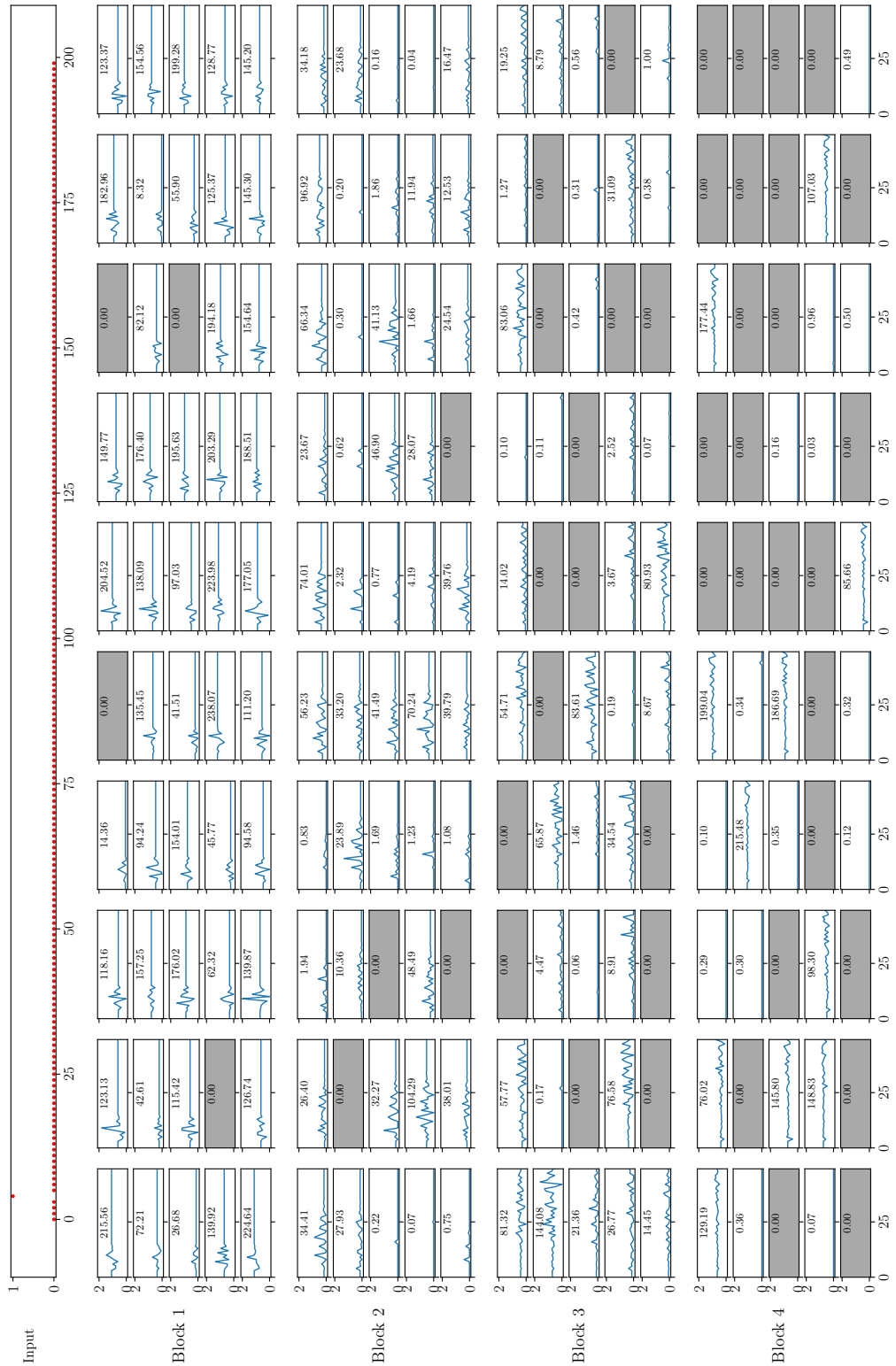


Figure B.2: Modification activation plots of a model trained on dataset A to a dirac function input signal.

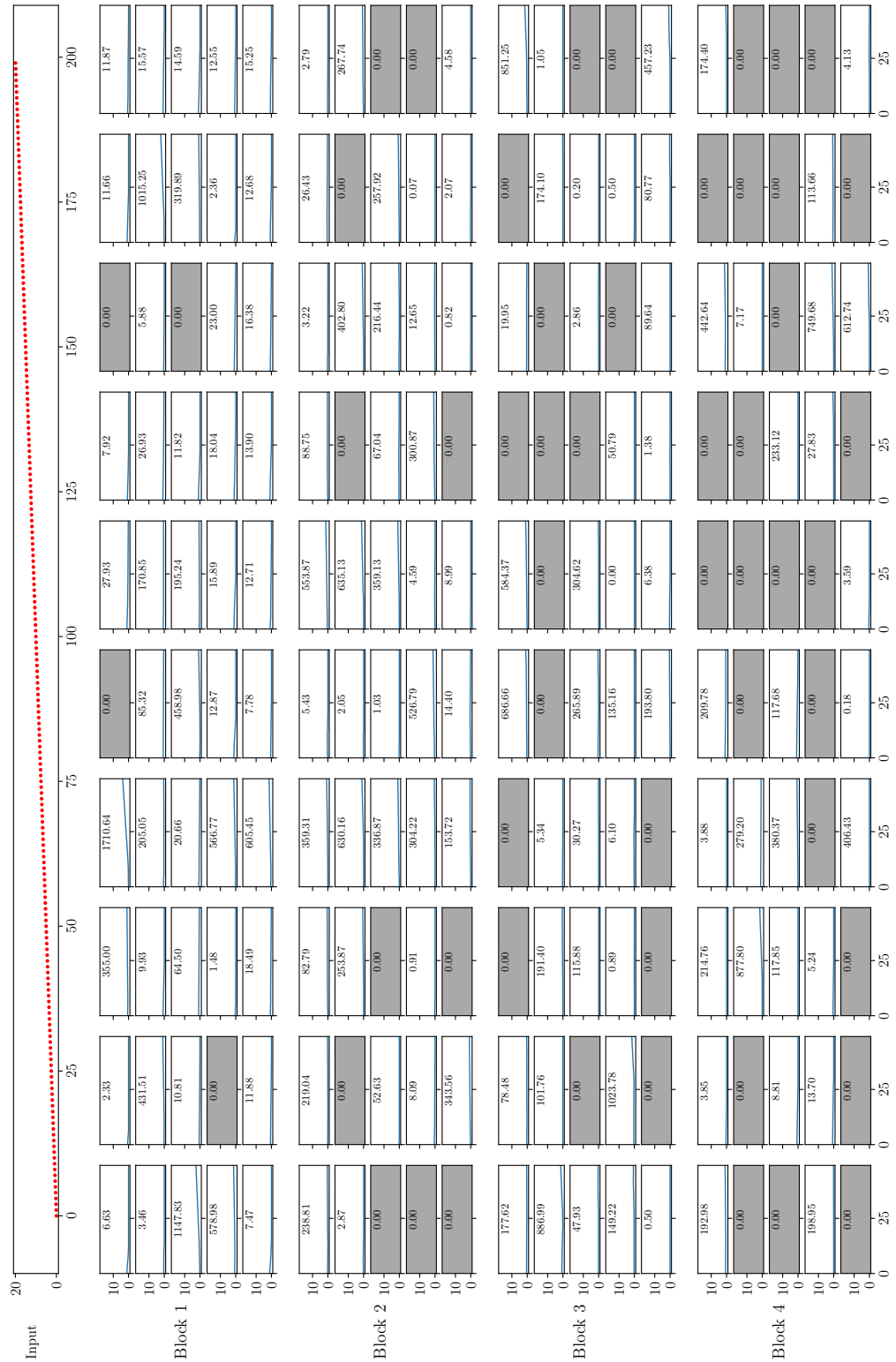


Figure B.3: Modification activation plots of a model trained on dataset A to a linear input signal.

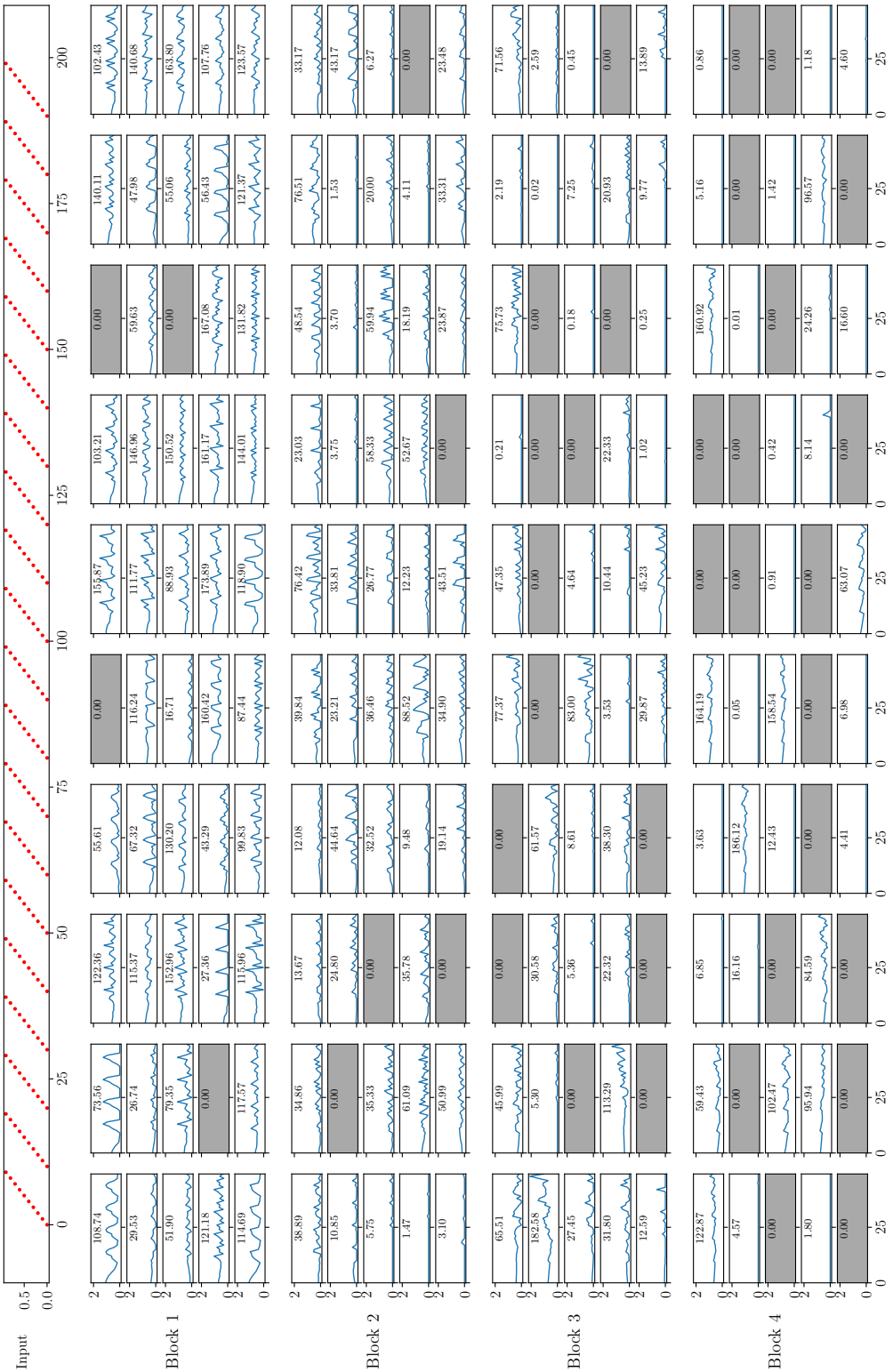


Figure B.4: Modification activation plots of a model trained on dataset A to a sawtooth input signal.

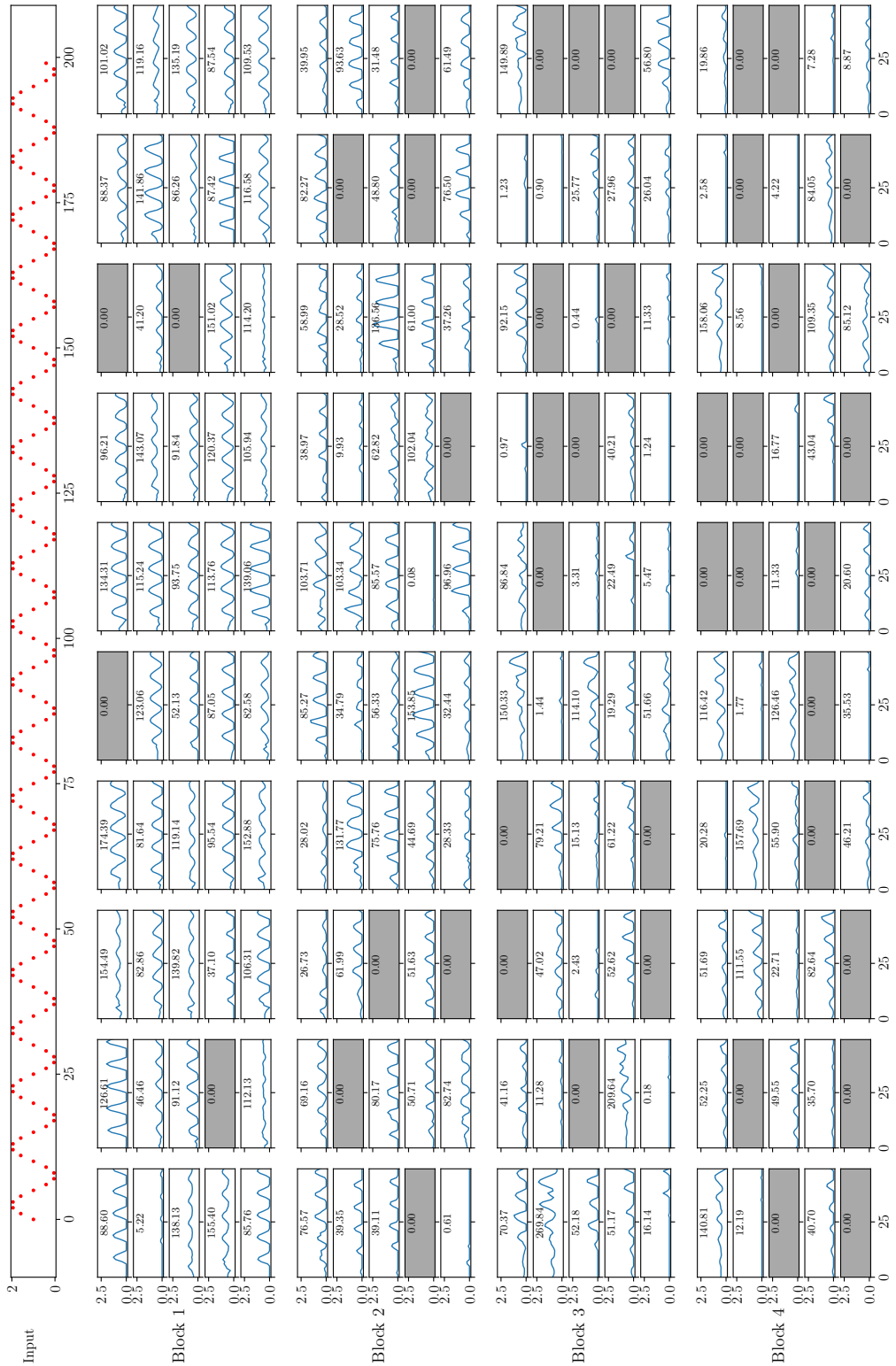


Figure B.5: Modification activation plots of a model trained on dataset A to a sine wave input signal.

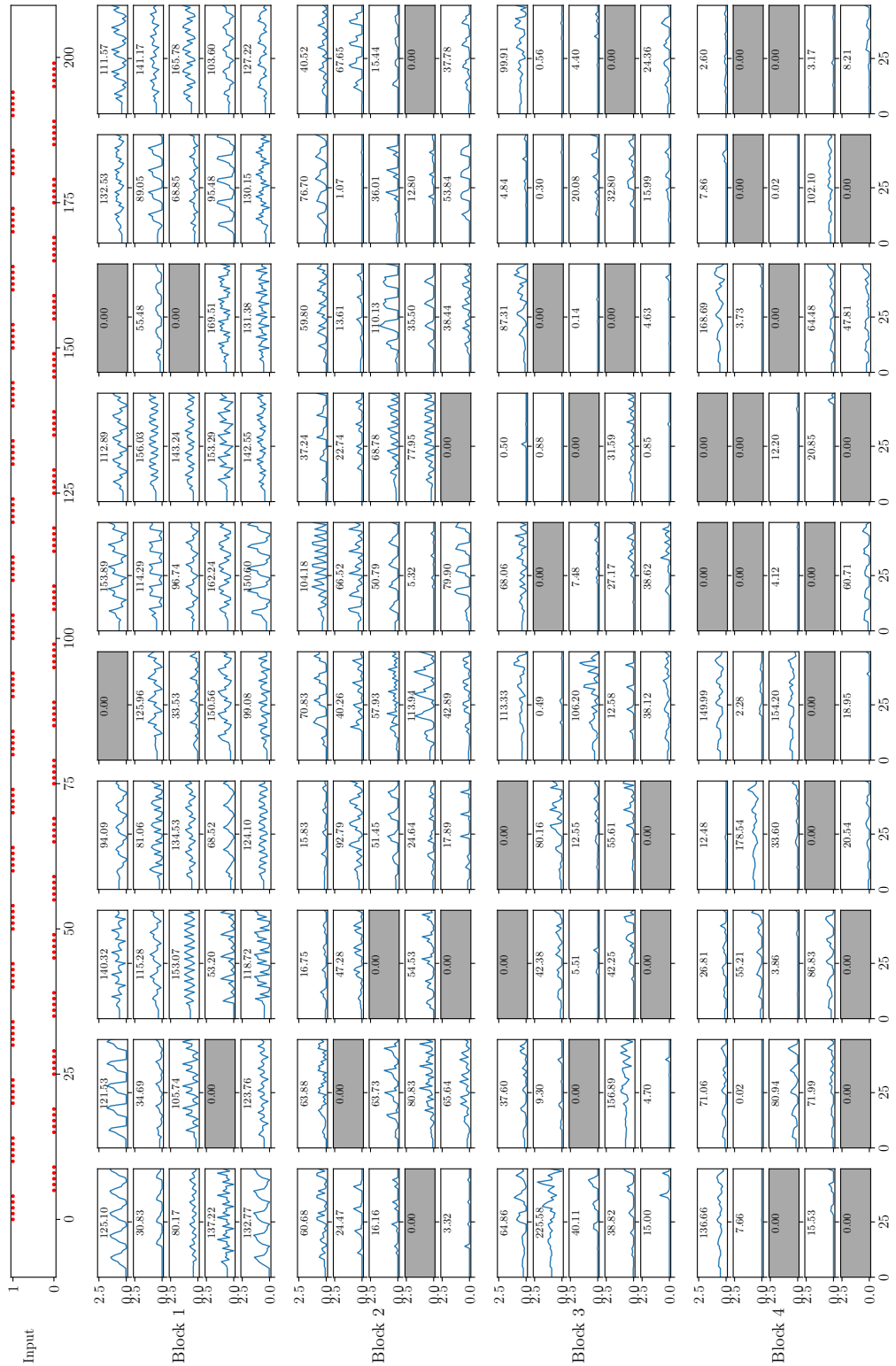


Figure B.6: Modification activation plots of a model trained on dataset A to a square wave input signal.

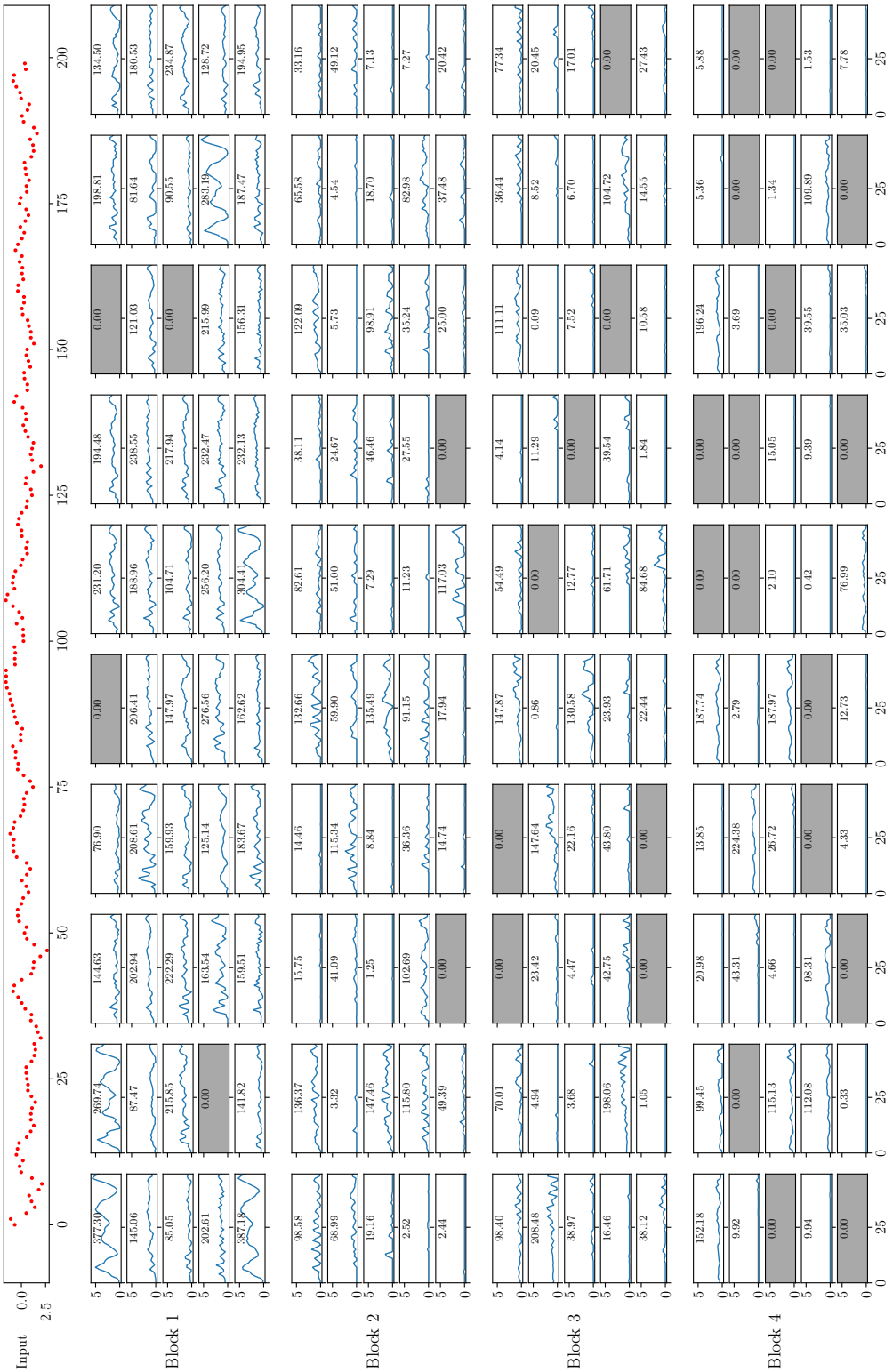


Figure B.7: Modification activation plots of a model trained on dataset A to a random test sequence input.

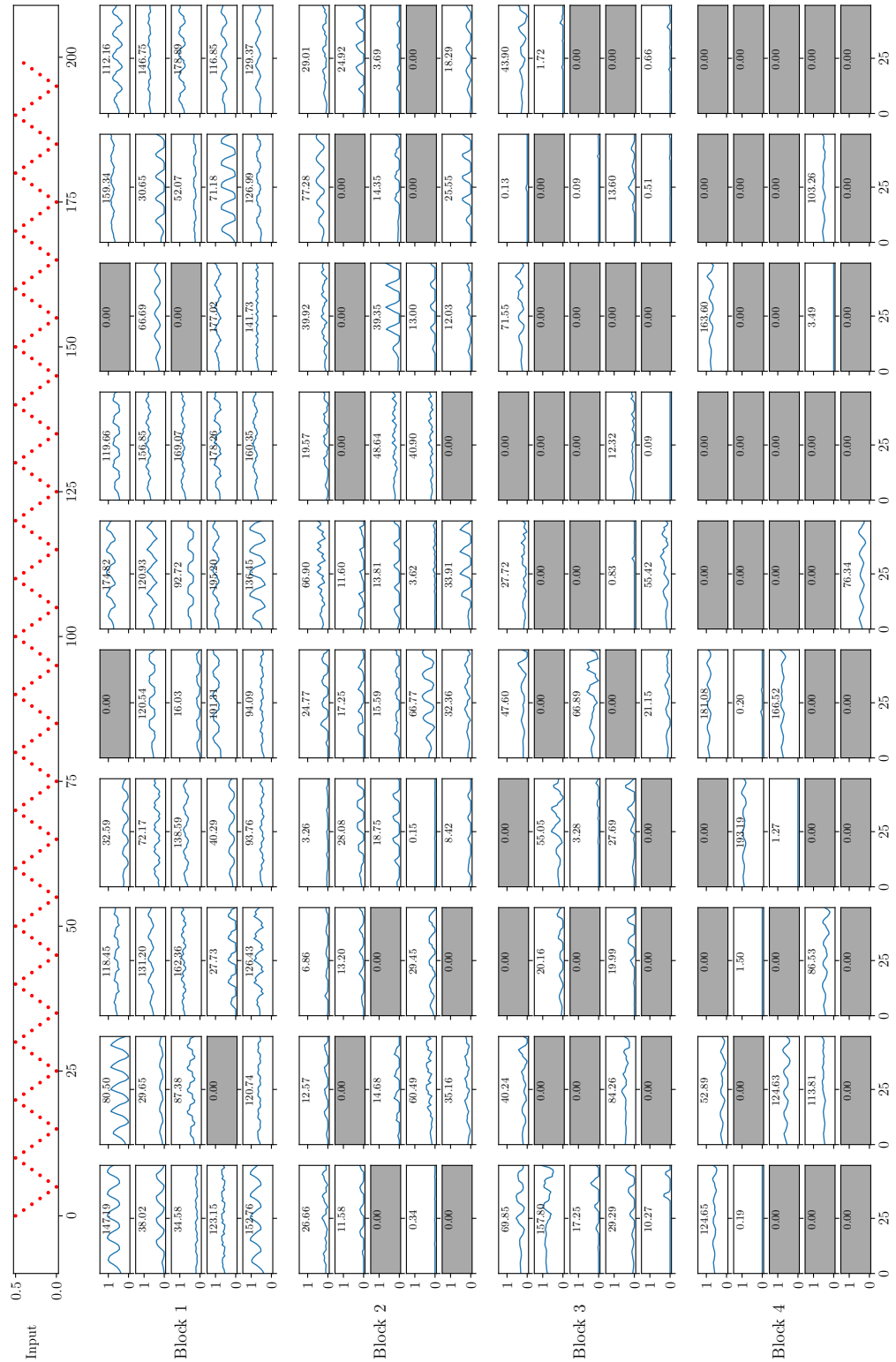


Figure B.8: Modification activation plots of a model trained on dataset A to a triangular wave input signal.

B.1.2 Dataset B

The following eight activation plots were obtained by exciting a TCN model trained on dataset B with the respective input sequences.

Figure B.9: Modification activation plots of a model trained on dataset B to a constant input signal.

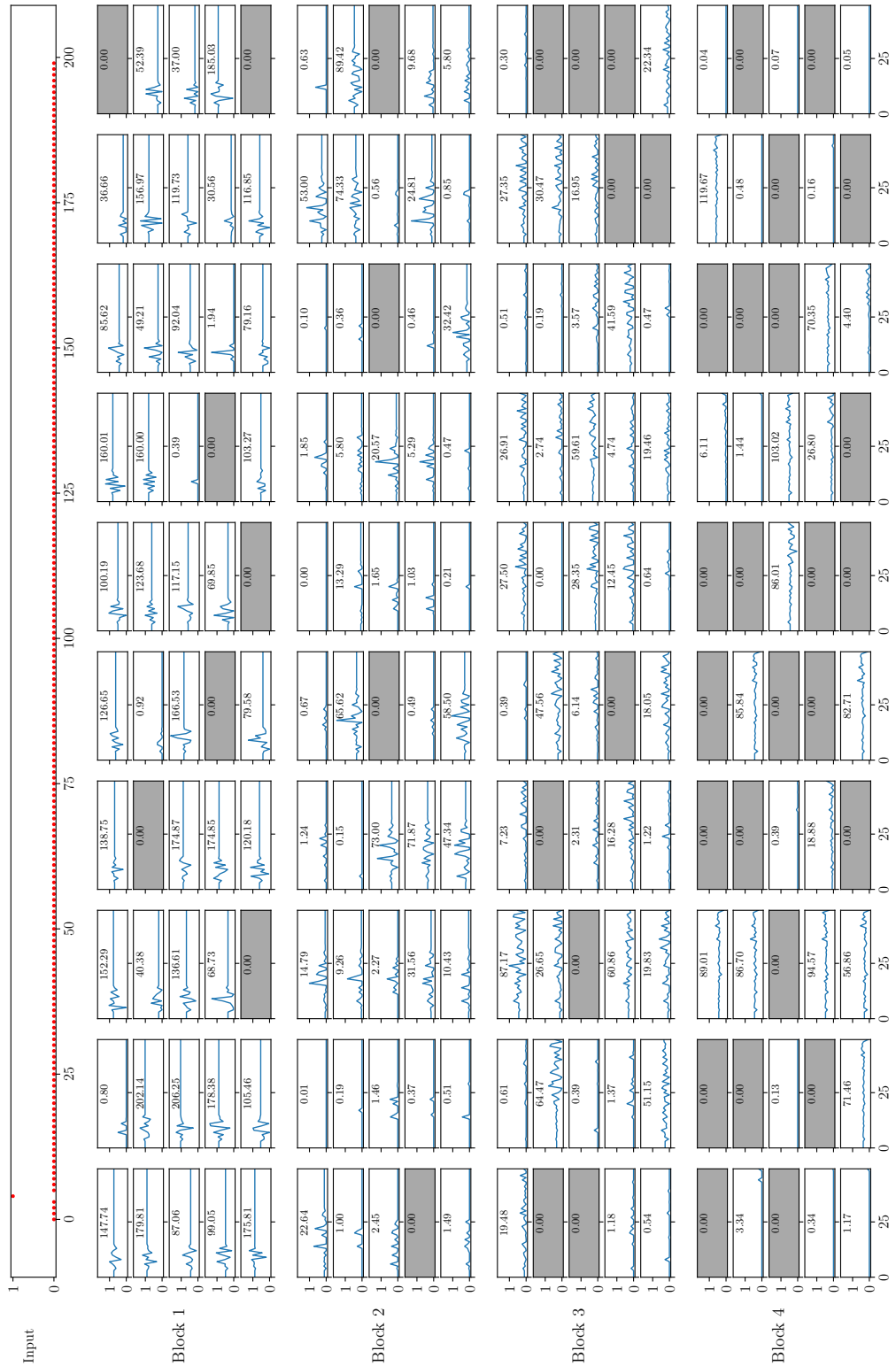


Figure B.10: Modification activation plots of a model trained on dataset B to a dirac function input signal.

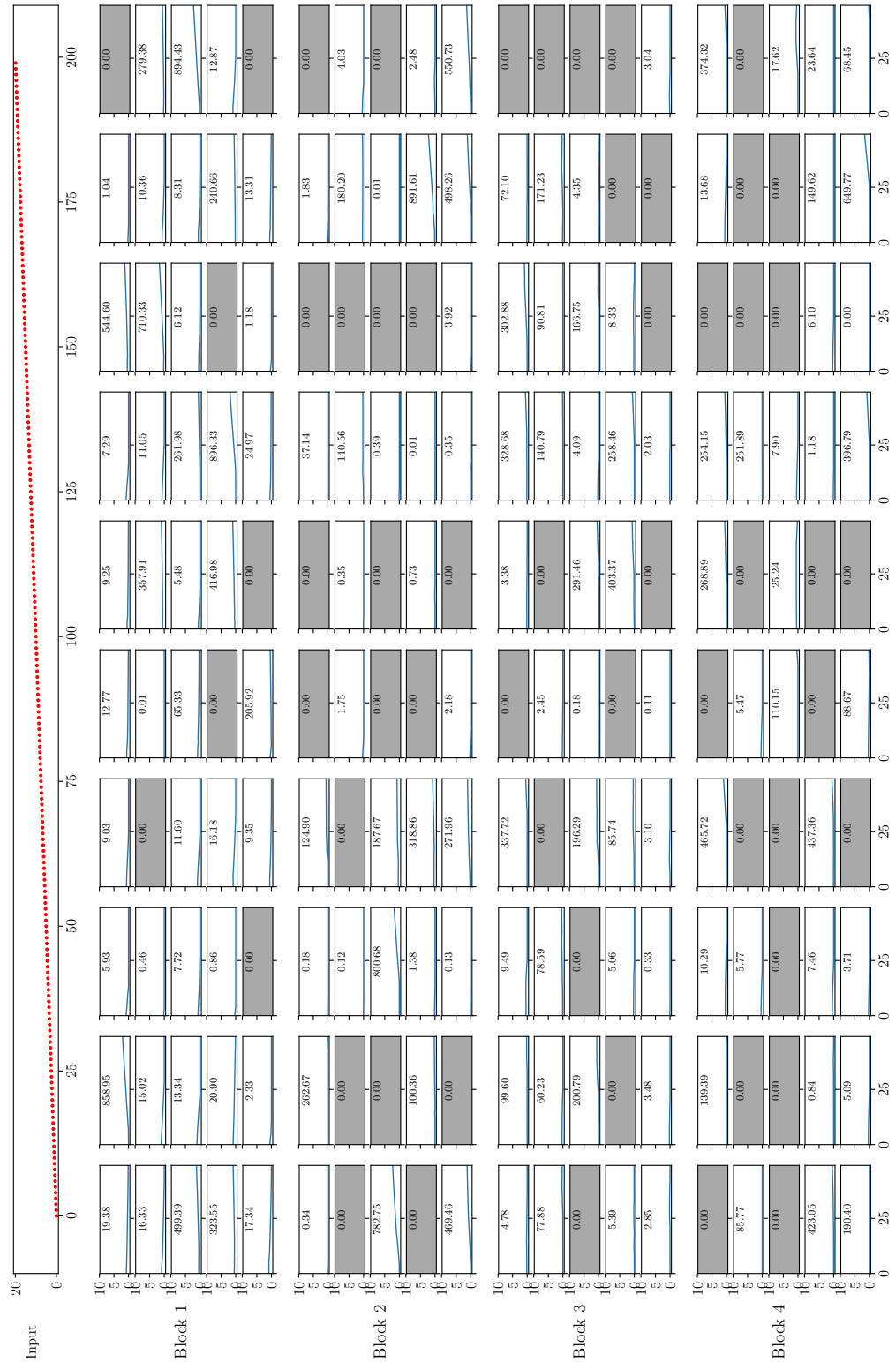


Figure B.11: Modification activation plots of a model trained on dataset B to a linear input signal.

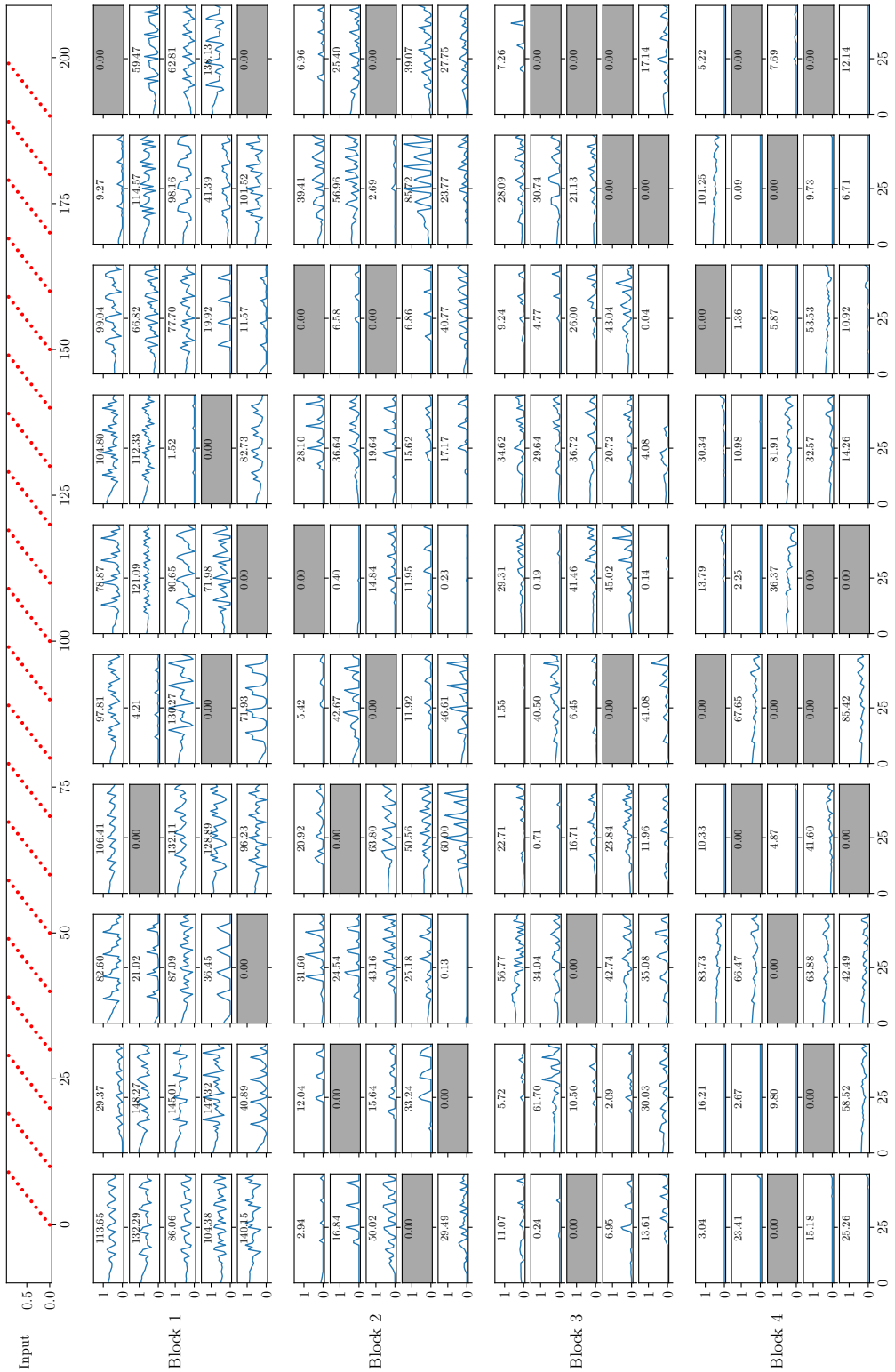


Figure B.12: Modification activation plots of a model trained on dataset B to a sawtooth input signal.

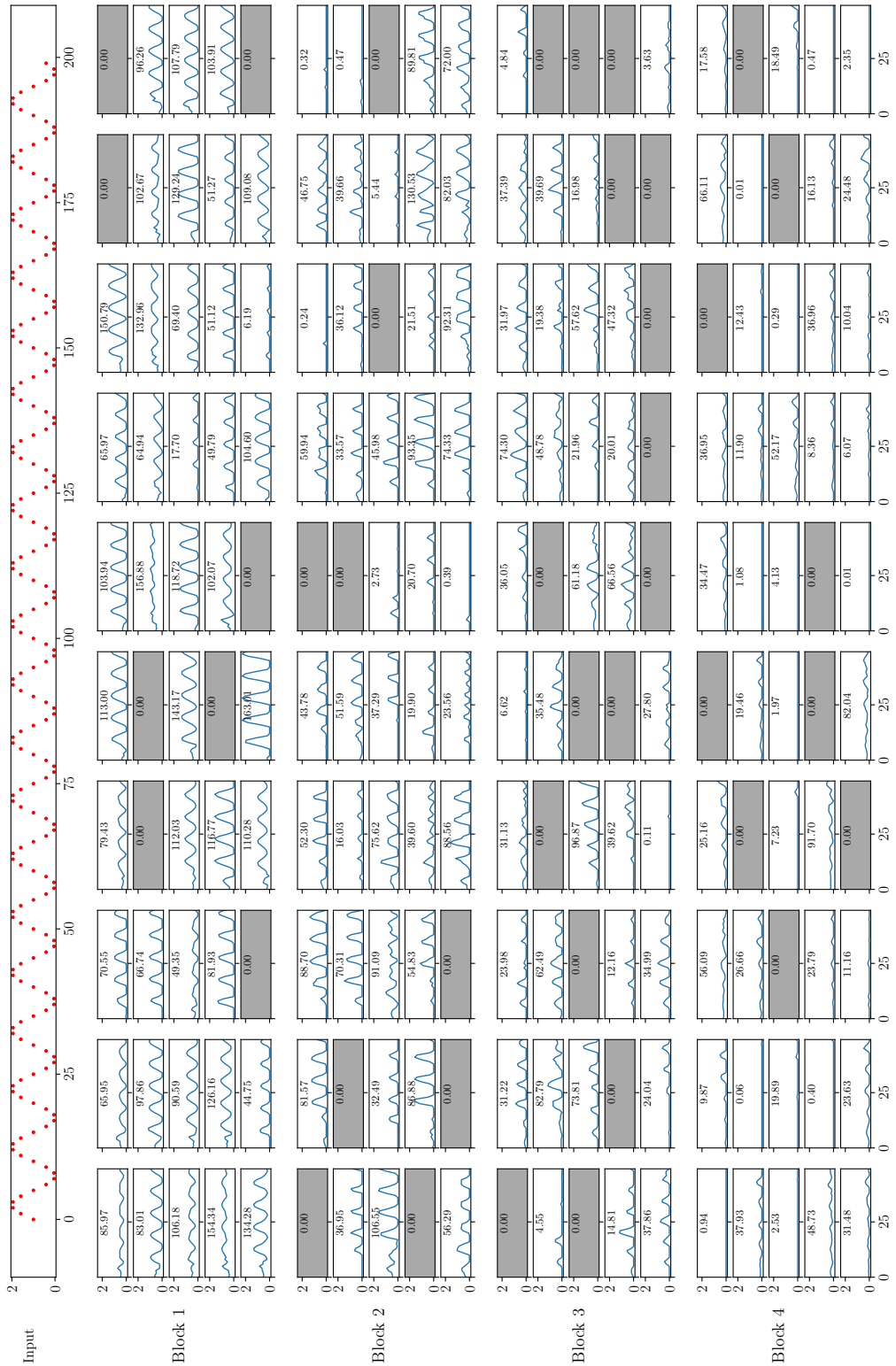


Figure B.13: Modification activation plots of a model trained on dataset B to a sine wave input signal.

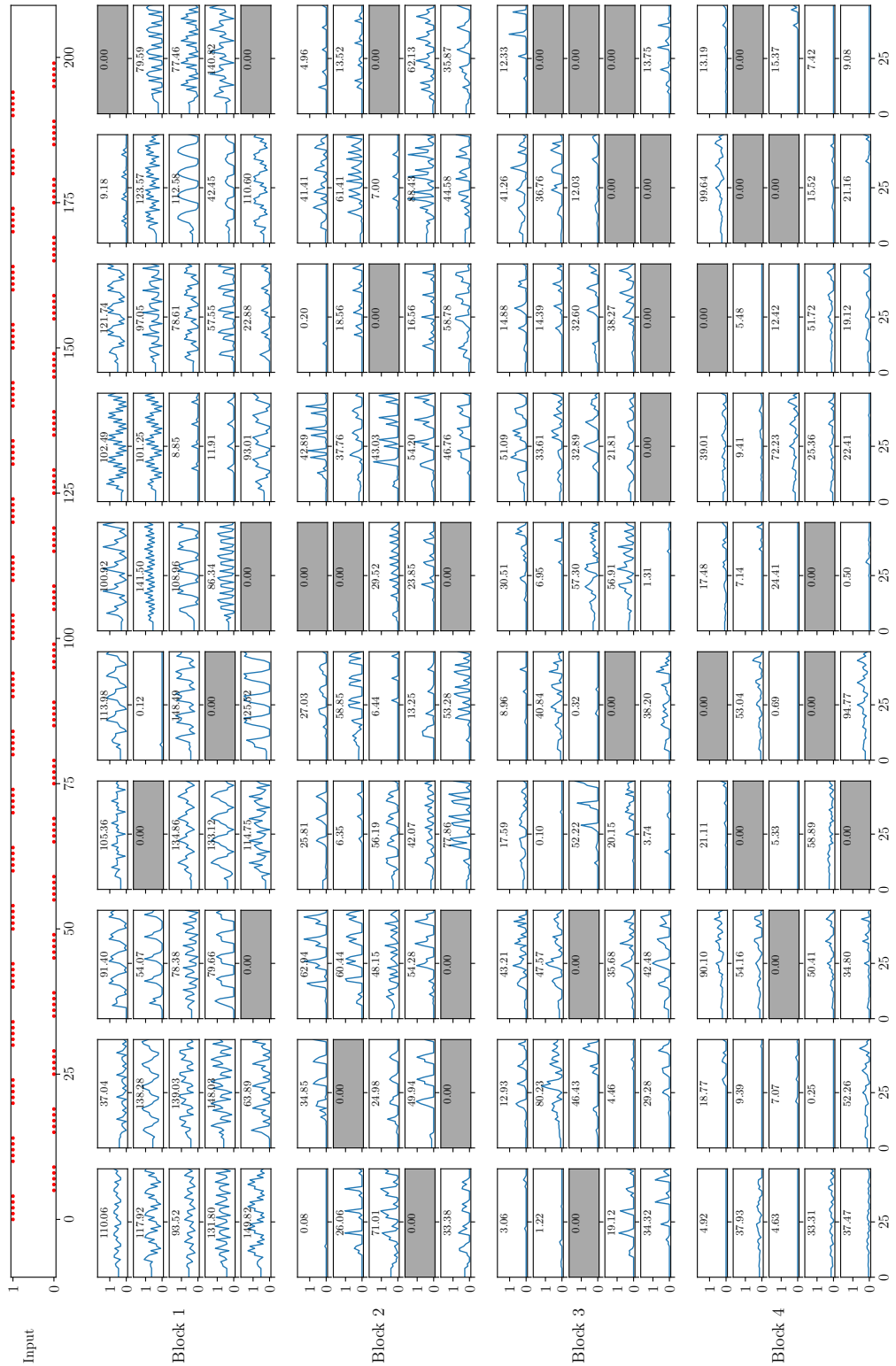


Figure B.14: Modification activation plots of a model trained on dataset B to a square wave input signal.

Figure B.15: Modification activation plots of a model trained on dataset B to a random test sequence input.

Figure B.16: Modification activation plots of a model trained on dataset B to a triangular wave input signal.

B.2 Horizontal Evaluation

Section 4.3 on the horizontal evaluation of layer knowledge evaluates the influence that a residual block has on the final output of the network, using a model trained on dataset B. In this appendix section, we provide the same results also for a model trained on dataset A (see figure B.17 and table B.1) which are qualitatively similar to the ones obtained for dataset B.

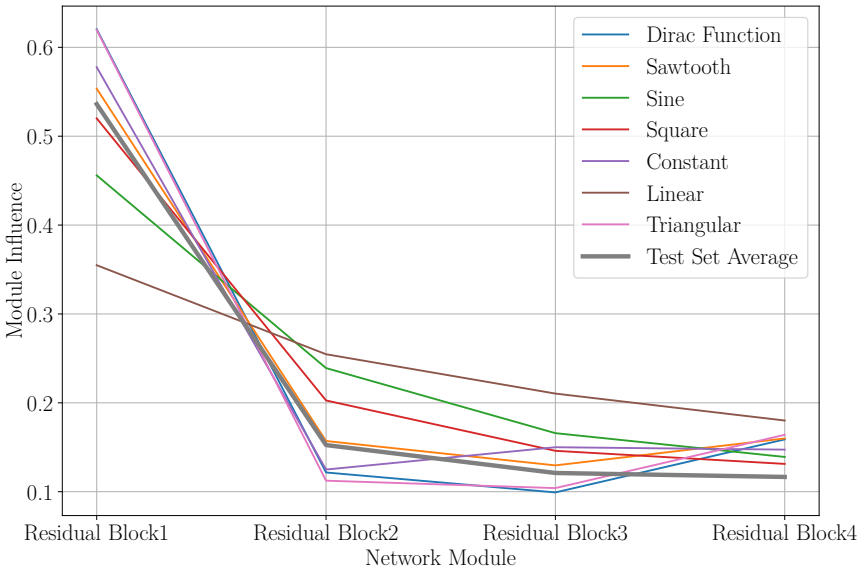


Figure B.17: Influence of residual blocks on the total activation of the network trained on dataset A. All modification curves except the *Test Set Average* result from a single input sequence of length 200. *Test Set Average* is the average influence of all sequences in the test set of A. The respective values can be observed in table B.1

	ResBlock 1	ResBlock 2	ResBlock 3	ResBlock 4
Dirac Function	62.0	12.2	9.9	15.9
Sawtooth	55.3	15.7	13.0	16.0
Sine	45.6	23.9	16.6	13.9
Square	52.0	20.3	14.6	13.1
Constant	57.8	12.5	15.0	14.7
Linear	35.5	25.5	21.0	18.0
Triangular	62.0	11.2	10.4	16.4
Test Set Average	53.6	15.2	12.1	11.7
Mean	53.0	17.0	14.0	15.0
Min	35.5	11.2	9.9	11.8
Max	62.0	25.5	21.0	18.0

Table B.1: Layer influence for different input signals of a TCN trained on dataset A. Influence values are calculated according to equation (4.5). *Test Set Average* denotes the average of all sequences in the test set of A. All values in percent of the output activation of the final residual block.

B.3 Vertical Evaluation

In section 4.4 on the vertical evaluation of layer knowledge, a quantification of the importance of individual channels in the network is presented. This quantification is based on a model trained on dataset B. In this appendix section, we provide the same results also for a model trained on dataset A (see figure B.18) which are qualitatively similar to the ones obtained for dataset B.

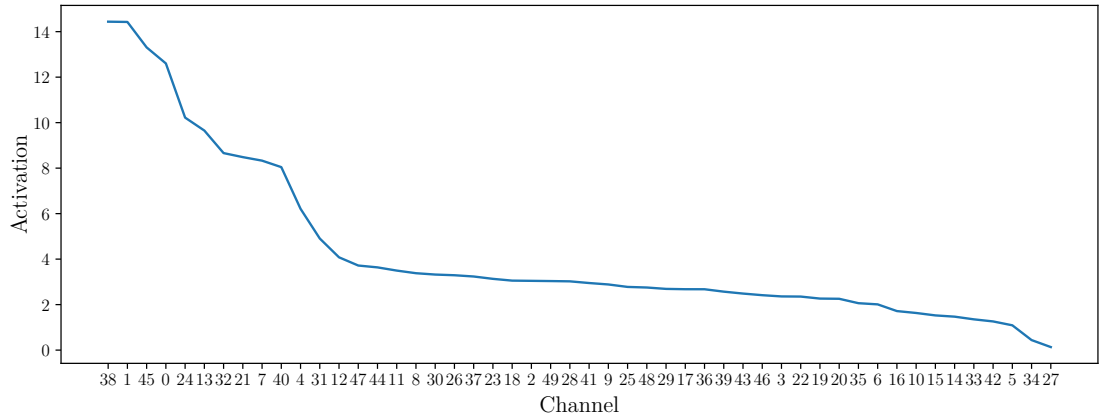


Figure B.18: Activation distribution of all fifty channels of the final network output. Activation values are the activations after the last residual block, scaled by the respective channel factors of the final dense layer of the network.