Impact of Time Series Data Augmentation on Latent Representation and Performance

Auswirkung von Data Augmentations bei Zeitreihen auf die latente Darstellung und Performanz

Bachelor thesis by Jonas Andreas Milkovits
Date of submission: January 22, 2024

1. Review: Prof. Dr. Kristian Kersting
2. Review: Maurice Kraus, M.Sc.
Darmstadt

TECHNISCHE UNIVERSITÄT DARMSTADT

Computer Science Department

Artificial Intelligence and Machine Learning Lab

Abstract

Time series are a fundamental component in various fields including finance, healthcare, engineering and economics. Recent studies have revealed that deep learning is a promising approach to tackle different tasks involving time series data. However, to perform well, deep learning models usually need lots of annotated data which leads to a major problem of working with time series as the difficult labeling process results in sparse labeled time series data. To combat this issue and enhance the size and quality of the training data, data augmentations can be used to generate synthetic views possibly covering unexplored input space. As there is still a lot to explore regarding data augmentations on time series, this thesis is performing an examination of the impact of various data augmentations on mainly two aspects. The first and more obvious aspect being investigated is the impact on the performance of the model. As only observing changes in performance metrics offers limited understanding of the modifications in neural networks (NNs) caused by data augmentations, a comprehensive analysis of the impact on latent representations is conducted.

In the pursuit of comprehending large neural networks, often considered as black boxes due to the interactions among numerous non-linear components, an interactive web application was developed to facilitate exploration. Prior to utilizing the web application, multiple models were trained with different data augmentations applied to their training datasets. The training datasets of these models consisted of different datasets, both univariate and multivariate time series.
To explore the features extracted by a specific convolutional layer, activation maximization (AM) was employed in conjunction with dimensionality reduction techniques to generate a map of all visual features, referred to as Activation Atlas. While the web application incorporates additional interactive visualizations to enhance the understanding of deep neural network functionality, a quantitative approach was also employed. This involved examining the models performance and network similarity, a metric introduced in this thesis.
Considering the significant challenge of understanding neural networks, particularly due to their depth, this thesis marks a meaningful step toward unraveling the complexities of these intricate systems.

Zusammenfassung

Zeitreihen sind eine grundlegende Komponente in verschiedenen Bereichen wie Finanzen, Gesundheitswesen, Technik und Wirtschaft. Jüngste Studien haben gezeigt, dass Deep Learning ein vielversprechender Ansatz zur Bewältigung verschiedener Aufgaben mit Zeitreihendaten ist. Um gute Ergebnisse zu erzielen, benötigen Deep Learning Modelle jedoch in der Regel viele gelabelte Daten. Dies führt zu einem großen Problem bei der Arbeit mit Zeitreihen, da der schwierige Labeling-Prozess zu einer geringen Anzahl an beschrifteten Zeitreihendaten führt. Zur Bekämpfung dieses Problems und zur Verbesserung der Qualität der Trainingsdaten können Data Augmentations verwendet werden, um synthetische Zeitreihen zu erzeugen, die möglicherweise einen unerforschten Eingaberaum abdecken. Da es in Bezug auf Data Augmentations bei Zeitreihen noch viel zu erforschen gibt, werden in dieser Arbeit die Auswirkungen verschiedener Data Augmentations auf hauptsächlich zwei Aspekte untersucht. Der erste Aspekt, der untersucht wird, sind die Auswirkungen auf die Vorhersagegenauigkeit des Modells. Da die bloße Beobachtung von Änderungen in den Leistungsmetriken nur ein begrenztes Verständnis der durch Data Augmentations verursachten Änderungen in neuronalen Netzen bietet, wird eine umfassende Analyse der Auswirkungen auf latente Repräsentationen durchgeführt.

In dem Bestreben, große neuronale Netze zu verstehen, die aufgrund der Wechselwirkungen zwischen zahlreichen nichtlinearen Komponenten oft als Blackboxen betrachtet werden, wurde eine interaktive Webanwendung entwickelt, um die Erkundung zu erleichtern. Vor dem Einsatz der Webanwendung wurden mehrere Modelle mit verschiedenen Data Augmentations für ihre Trainingsdatensätze trainiert. Die Trainingsdatensätze dieser Modelle bestanden aus verschiedenen Datensätzen, sowohl univariaten als auch multivariaten Zeitreihen.
Um die von einer bestimmten Faltungsschicht extrahierten Merkmale zu untersuchen, wurde Activation Maximization in Verbindung mit Dimensionalitätsreduktionstechniken eingesetzt, um eine Karte aller Merkmale, die das Modell extrahiert, zu erstellen, die als Aktivierungsatlas bezeichnet wird. Während die Webanwendung zusätzliche interaktive Visualisierungen enthält, wurde zusätzlich ein quantitativer Ansatz gewählt, um das Verständnis der Funktionalität tiefer neuronaler Netze zu verbessern. Dazu wurden die Leistung der Modelle und die Ähnlichkeit der Netzwerke untersucht. In Anbetracht der großen Herausforderung, neuronale Netze zu verstehen, insbesondere aufgrund ihrer Tiefe, stellt diese Arbeit einen bedeutenden Schritt zur Entschlüsselung der Komplexität dieser komplizierten Systeme da.

Erklärung zur Abschlussarbeit gemäß § 22 Abs. 7 APB TU Darmstadt

Hiermit erkläre ich, Jonas Andreas Milkovits, dass ich die vorliegende Arbeit gemäß § 22 Abs. 7 APB der TU Darmstadt selbstständig, ohne Hilfe Dritter und nur mit den angegebenen Quellen und Hilfsmitteln angefertigt habe. Ich habe mit Ausnahme der zitierten Literatur und anderer in der Arbeit genannter Quellen keine fremden Hilfsmittel benutzt. Die von mir bei der Anfertigung dieser wissenschaftlichen Arbeit wörtlich oder inhaltlich benutzte Literatur und alle anderen Quellen habe ich im Text deutlich gekennzeichnet und gesondert aufgeführt. Dies gilt auch für Quellen oder Hilfsmittel aus dem Internet.

Diese Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.

Mir ist bekannt, dass im Falle eines Plagiats (§ 38 Abs. 2 APB) ein Täuschungsversuch vorliegt, der dazu führt, dass die Arbeit mit 5,0 bewertet und damit ein Prüfungsversuch verbraucht wird. Abschlussarbeiten dürfen nur einmal wiederholt werden.

Bei einer Thesis des Fachbereichs Architektur entspricht die eingereichte elektronische Fassung dem vorgestellten Modell und den vorgelegten Plänen.

Darmstadt, 22. Januar 2024

J. Milkovits

Contents

List of Figures

List of Tables

List of Abbreviations

CNN convolutional neural network

FCN fully convolutional neural network

RNN recurrent neural network

MLP multi-layer perceptron

GPU graphics processing unit

STL seasonal and trend decomposition using locally estimated scatterplot smoothing

ReLU rectified linear unit

GAN generative adversarial networks

SGD stochastic gradient descent

AM activation maximization

CAM class activation map

EAP extreme activation penalty

PCA principal component analysis

DSADS daily and sports activities

NN neural network

1 Introduction

1.1 Motivation

Time series are a fundamental component in various fields including finance, healthcare, engineering and economics. This type of data is extremely relevant, serving purposes such as anomaly detection, classifying samples and forecasting future trends. Recent studies have revealed that deep learning is a promising approach to tackle different tasks involving time series data [5, 22, 46]. However, to perform well, deep learning models usually need lots of annotated data which leads to a major problem of working with time series as the difficult labeling process results in sparse labeled time series data. To combat this issue and enhance the size and quality of the training data, data augmentations can be used to generate synthetic views possibly covering unexplored input space. Data augmentations exist in various forms and can also be used as a basis for contrastive methods in self-supervised pre-training [36, 23].
As there is still a lot to explore regarding data augmentations on time series, this thesis is performing an examination of the impact of various data augmentations on mainly two aspects. Firstly, this thesis investigates the impact on the performance of the model. Many studies predominantly compare data augmentations by assessing changes in performance metrics [23, 39]. This results in a limited understanding of modifications in NNs caused by data augmentations. This thesis aims for a deeper understanding and therefore a comprehensive analysis of the impact on the latent representation is conducted. To enhance the results' exploration and comprehension this thesis includes an interactive web-based visualization of the results. This contribution simplifies the process of comprehending alterations within NNs by facilitating almost effortless interaction with these networks. It paves the way for a more meaningful and well-founded exploration of data augmentations, convolutional neural networks (CNNs) and time series analyses.

1.2 Structure of the Thesis

This thesis is organized into distinct chapters, each focusing on different aspects. First the motivation for this research is presented. Afterwards various theoretical background elements are explained in detail, including time series, CNN, data augmentation techniques, visualization methods and datasets utilized.
The methodology chapter introduces tools such as TimeLens, TimeLens-Web and addresses aspects

of reproducibility and metrics used. Subsequently, the experimental setup and results are presented, covering activation maximization, data augmentations and providing both qualitative and quantitative analyses.
Research findings are discussed in the following chapter. The results are combined and interpreted, exploring the implications of data augmentations on time series data. Finally, this thesis concludes with a summary of key takeaways, methodological limitations and suggestions for future research directions.

2 Background

This background section describes the theoretical background for the following explorations. Specifically it provides an introduction to the intricacies of time series, the architecture and training procedure of CNNs, data augmentations, and finally, addressing visualization techniques.

2.1 Time Series

A time series is an ordered set of values [22]. One specific observation of a time series is called realisation of the time series, which are either observed at equally or unequally distanced points of time [20]. As stated by Bee Dagum and Bianconcini [4], the most important property of a time series is that the ordered observations are dependent through time. However, it should be noted that time does not have to represent the actual time and is often just used to represent the sequence order [23]. While univariate time series only observe a single variable over a certain time frame, multivariate time series observe multiple values and therefore consist of multiple time series, often with the same length and sampling rate. An example for a multivariate time series can be seen in figure 2.1, where the acceleration of two body limbs are recorded in x, y and z direction. All measurements share the same time axis and the observations are equally spaced in time.

Time series are very diverse and are used in different scientific disciplines such as economics with the daily fluctuations of the stock market. Social sciences might use them to evaluate countries based on their birthrates or school enrollments. Medicine also uses them frequently when performing magnetic resonance imaging of brain-wave time series patterns or to trace blood pressure measurements over time. Another example for a field which uses many multivariate time series is human activity recognition trying to predict a certain movement of a person by capturing the acceleration and orientation of sensors attached to their bodies [20].

The following mathematical notation for time series will be used in the following: Let X be the multivariate time series with dimensions (c, t), where c is the number of channels and t is the length of the time series. For simplicity purposes, it will be assumed that the length of the time series is the

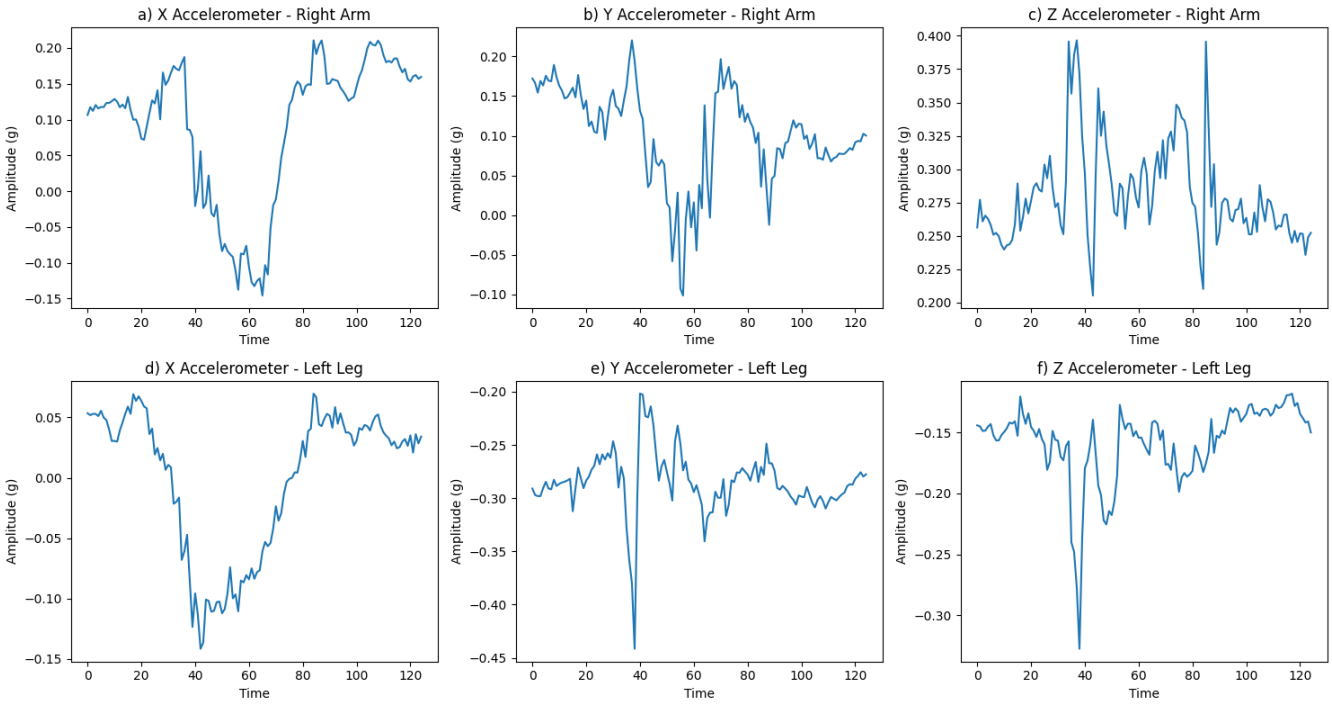


Figure 2.1: Multivariate time series sample from the DSADS dataset from a person performing rowing. All measurements share the same time axis. Each measurement records the acceleration in a specific direction for a certain limb. a) acceleration in x direction of the right arm b) acceleration in y direction of the right arm c) acceleration in z direction of the right arm d) acceleration in x direction of the left leg e) acceleration in y direction of the left leg f) acceleration in z direction of the left leg

same for each channel. The matrix can therefore be expressed as:

$$X = \begin{bmatrix} x_{1,1} & x_{1,2} & \cdots & x_{1,t} \\ x_{2,1} & x_{2,2} & \cdots & x_{2,t} \\ \vdots & \vdots & \ddots & \vdots \\ x_{c,1} & x_{c,2} & \cdots & x_{c,t} \end{bmatrix} \tag{2.1}$$

$x_{i,j}$ represents the realisation of channel i at time step j of the original time series. In summary each row of the matrix represents one time series at a certain channel.
In case of an univariate time series ($c = 1$), our matrix X would instead be the following vector:

$$X = \begin{bmatrix} x_{1,1} & x_{1,2} & \cdots & x_{1,t} \end{bmatrix} \tag{2.2}$$

which can be simplified to:

$$X = \begin{bmatrix} x_1 & x_2 & \cdots & x_t \end{bmatrix} \tag{2.3}$$

2.1.1 Time Series Patterns and Decompositions

To gain deeper insights into time series, it is possible to decompose them into sub-series. Understanding these decompositions requires a prior familiarity with the patterns of time series. A trend exists when there is a long-term increase or decrease in the time series data, which doesn't have to be linear. Another pattern is seasonality which describes the impact of seasonal factors such as the time of the year on the time series. Seasonal patterns are always of a fixed length and known frequency. Cycles on the other hand are not of a fixed frequency like seasonality. One example might be business cycles which repeat every few years (e.g. figure 2.2). Another example, figure 2.3, does not contain evidence for any cyclic behavior, but shows a significant rising trend with strong yearly seasonality [20]. Iwana and Uchida [23] state that decomposition methods in general decompose time series signals by extracting features or underlying patterns. One potential use case for decompositions is the independent usage or combination of those features for data augmentation.

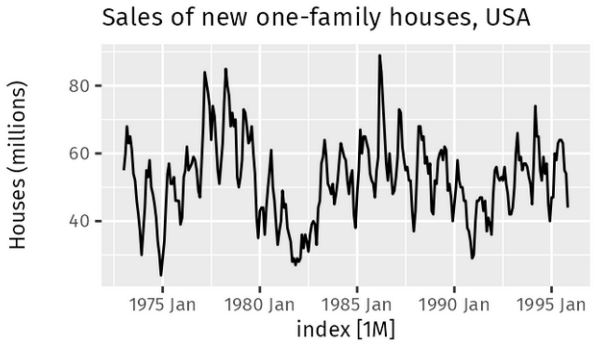


Figure 2.2: Sales of new one-family houses, USA - Seasonality and Cycles [20]. The time series contains cyclic behavior.

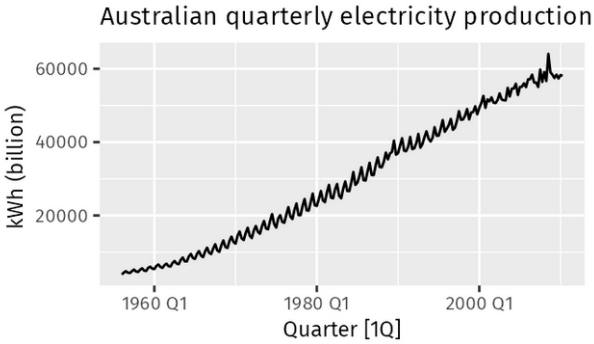


Figure 2.3: Australian electricity production - Trend and Seasonality [20]. The time series exhibits both trend and seasonality.

The classical decomposition method is either performed additive or multiplicative. It forms the basis for many other decomposition methods like X11 [4] and seasonal extraction in ARIMA time series [14]. The classical method assumes a constant seasonal component. This assumption might be valid for shorter time series, but is often flawed for longer time series where patterns change over the years [20].
Additive decomposition for an univariate time series defined in equation 2.3 can be formulated as

$$x_t = S_t + T_t + R_t \tag{2.4}$$

where x_t is the time series data point at period t, S_t is the seasonal component at period t, T_t is the combination of trend and cycle into a single trend-cycle component at period t and R_t is the remainder component at period t. Multiplicative decomposition on the other hand can be written as:

$$x_t = S_t \times T_t \times R_t \tag{2.5}$$

To perform additive decomposition, the first step is to compute the trend-cycle component by using a moving average paired with the seasonal period m. The weighted moving average can computed with the following formula where the weights are depending on whether m is an even or odd number to ensure that the calculation is symmetric.

$$\hat{T}_t = \sum_{j=-k}^{k} a_j y_{t+j} \tag{2.6}$$

Next, the detrended time series can be computed by subtracting the estimated trend from the data: $x_t - \hat{T}_t$. Estimation for the season component for each season is performed by averaging the detrended values for that season. Afterwards the seasonal component values are adjusted to guarantee that their sum equals to zero.
To finally calculate the remainder component, the estimated seasonal and trend-cycle component is subtracted from the data.

$$\hat{R}_t = x_t - \hat{T}_t - \hat{S}_t \tag{2.7}$$

The derivation for multiplicative decomposition is similar. However, instead of using subtractions the multiplicative decomposition works with divisions [20].
An example for additive decomposition on a sample time series can be seen in figure 2.4.

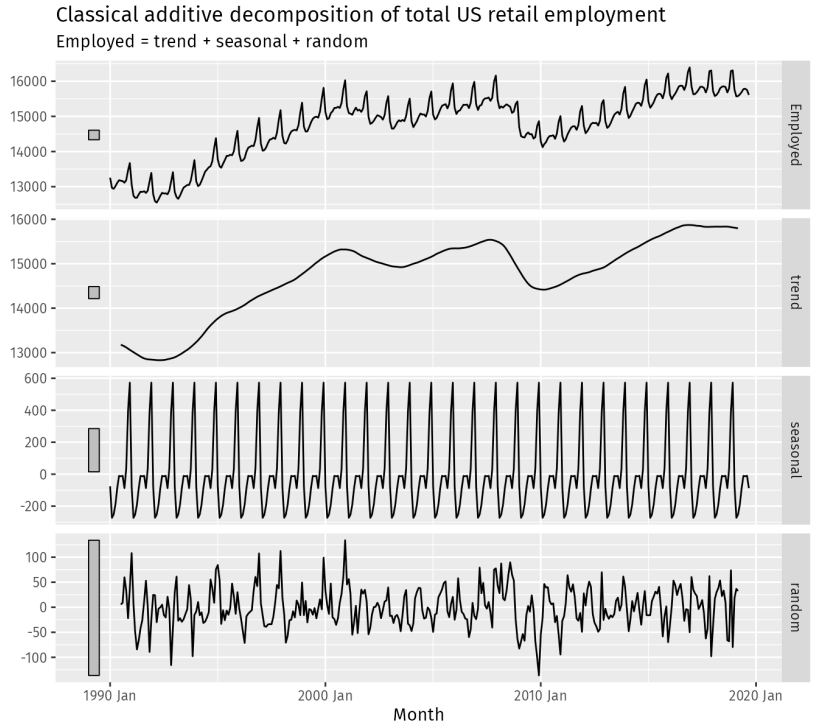


Figure 2.4: Classical additive decomposition applied to a time series of US retail employment. The top time series depicts the original time series. Displayed below are the extracted trend, seasonal and remainder component displayed as distinct time series.

While classical decomposition is still widely used, it is not recommended as there are better performing methods like seasonal and trend decomposition using locally estimated scatterplot smoothing (STL)

[20, 11]. One advantage compared to the classical method is that the seasonal component is allowed to change over time. This enables a better estimation accuracy for the seasonal component when decomposing longer time series. Furthermore, STL lets the user control certain parameters allowing them to adjust the method to their specific dataset. The two main parameters are the trend-cycle window and the seasonal window. As these both control how quickly the trend-cycle and seasonal components can change, the usage of smaller values leads to quicker changes in the components. An example for STL Decomposition can be seen in figure 2.5.

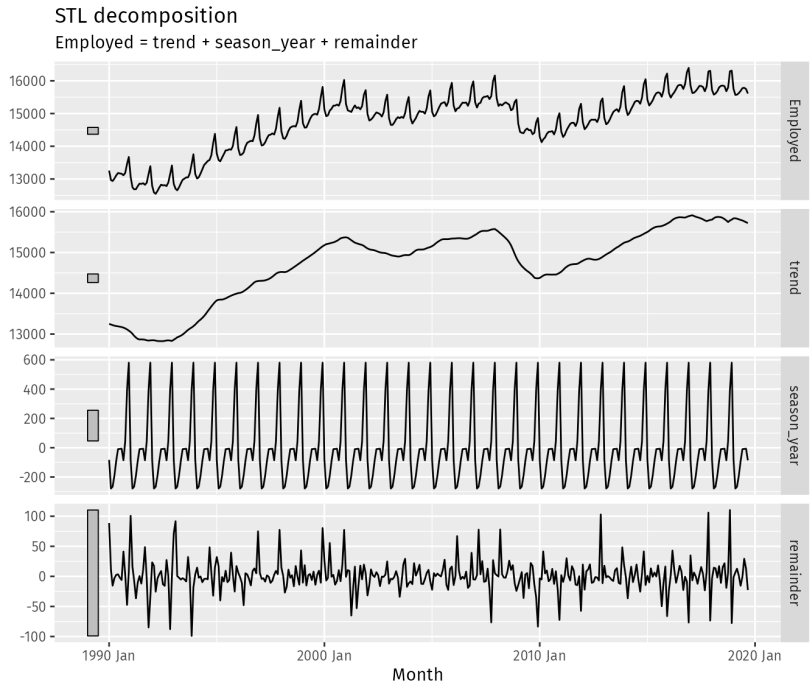


Figure 2.5: STL performed on a time series of US retail employment. The top time series depicts the original time series. Displayed below are the extracted trend, seasonal and remainder component displayed as distinct time series.

As mentioned before, the features extracted by decomposition might be used for data augmentation. Moreover, these features can be used to measure the strength of trend and seasonality in a time series by using equation 2.8 and 2.9 [45]. These measures might prove useful when looking for time series in a dataset with a strong trend or seasonality [20]. To measure the strength of trend, the following formula can be used:

$$F_T = \max(0, 1 - \frac{Var(R_t)}{Var(T_t + R_t)} \tag{2.8}$$

The formula assumes that strongly trended data leads to higher variation in the seasonally adjusted data than the remainder component. On the other hand, the variances should be roughly equal for weakly trended data.
To measure the strength of seasonality, a similar formula can be used:

$$F_T = \max(0, 1 - \frac{Var(R_t)}{Var(S_t + R_t)} \tag{2.9}$$

2.1.2 Time Series Tasks

There are several goals when analyzing time series data. In time series classification a dataset of labeled time series is used to train a classifier to assign a label to an unseen sample. The dataset in this case could be denoted as $\{(X_1, y_1), \ldots, (X_N, y_N)\}$ with X_t being the time series and y the corresponding label for that sample.
A task that usually heavily makes use of time series patterns and decompositions as described in section 2.1.1 is time series forecasting. It aims to predict the future behavior by analysing dynamics of historical temporal data. If $X = [x_1, \ldots, x_T]$ are the historical observations, the goal is to predict the future values $[x_{T+1}, x_{T+2}, \ldots]$.
Another method is time series clustering, where the objective is the partitioning of the time series dataset into a certain amount of clusters K. Clustering tries to maximize similarities between samples from the sample cluster and dissimilarities from samples of different clusters. As clustering is an unsupervised method, it can be utilized on massive, unlabeled datasets.
Some time series contain samples that deviate from other samples in a certain way. One example is the measurement of seismic activities and detecting suspicious readings. Time series anomaly detection aims to identify those observations in the time series [29].

2.2 Convolutional Neural Networks

CNNs are a special type of neural network and are widely known for their breakthroughs in computer vision. Two of the most influential networks are LeNet [27] and AlexNet [26]. LeNet-5 popularized the idea of CNNs by demonstrating the effectiveness of using convolution layers in image classification tasks, in this case handwritten digit recognition in 1998. AlexNet was created several years later and was the first NN to win the yearly ImageNet [13] large-scale visual recognition challenge. Leveraging the power of training in parallel using multiple graphics processing units (GPUs), they were able to train a deeper network compared to LeNet achieving a dramatic performance improvement. The deepness of a neural network refers to the amount of trained layers. Demonstrating the power of deep NNs spiked interest and confidence in deep learning and CNNs leading to a significant increase in the usage of deep convolutional architectures [43]. With CNNs being a popular architecture up to this day, this thesis explores the inner workings and structure of CNNs while also discussing how the training and optimization processes function in the following sections. Even though CNNs are known for their huge part in computer vision, the following sections will explain the underlying concepts referring to time series instead of images as the intuition is similar.

2.2.1 Architecture and Operation

CNNs typically consist of two parts. The initial segment often comprises an alternation of convolutional layers and pooling operations, aiming to extract useful features for the downstream task from

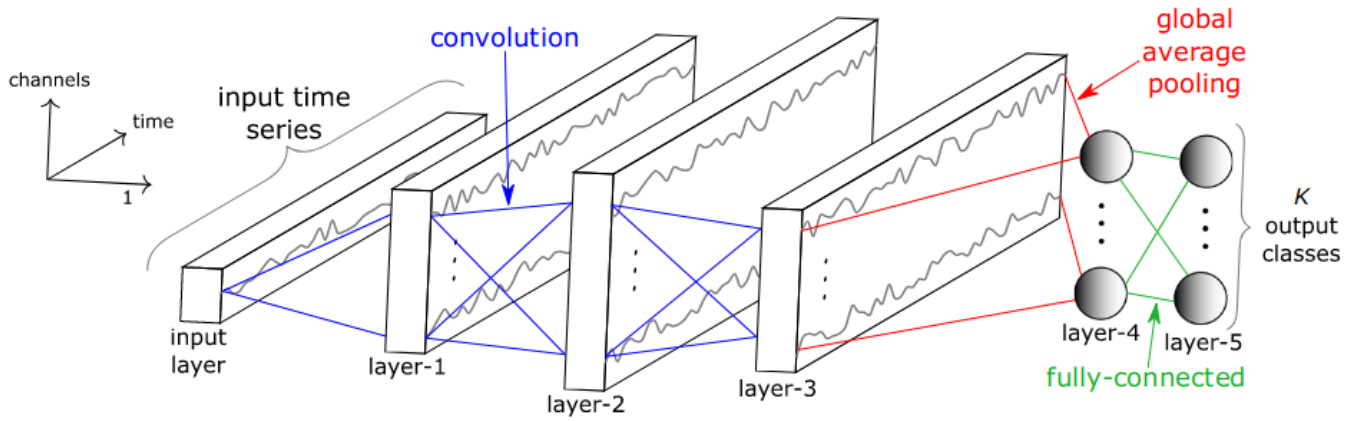


Figure 2.6: Exemplary CNN architecture for time series. Three convolutional layers and two fully-connected layers are shown. The input layer consists of a time series with potentially multiple channels. [22]

the training data [50, 37]. The second part is usually a fully-connected layer, but might be omitted in fully convolutional neural networks (FCNs). Convolution layers use convolutions to transform the input into so-called feature maps, which can be described as many channels stacked next to each other. The convolution process consists of performing the dot product of the filter and a small region of the input and replicating this step across the entire input. The convolution process can be formulated by

$$C_t = f(\omega \cdot X_{t-\frac{l}{2}:t+\frac{l}{2}} + b) | \forall t \in [0, T] \tag{2.10}$$

where the dot product of a filter ω of length l applied on a part of a time series X results in the convolution C [22]. A non-linear activation function f, often the rectified linear unit (ReLU) [33] function, is applied on the output of the dot product [43].

For the first convolutional layer the input is the measured time series. Any subsequent convolutional layer takes the feature maps of the previous layer as an input, transforms them into their feature maps and passes these into the next convolutional layer. Performing a convolution on time series uses a 1D kernel of length n with the following notation:

$$k = \begin{bmatrix} k_1 & k_2 & \cdots & k_n \end{bmatrix} \tag{2.11}$$

Instead of a single value stored in a node like in multi-layer perceptrons (MLPs), the trainable weights of CNNs are stored in the kernels. However, the shape of the output doesn't solely rely on the size of the kernel, but also on the stride and padding parameter of the convolution. While padding controls the amount of values added to the input to counter the reduction in length caused by the convolution operation, stride adjusts how much the kernel moves after performing the dot product of the time series values at the current position and the kernel itself. To further help the comprehension of the convolution process for time series, figure 2.7 can be examined. It visualizes the process of 1D convolution with a given kernel of size five on a padded signal.

To calculate the size of the output of a convolution, the following formula can be used:

$$\frac{W - K + 2P}{S} + 1 \tag{2.12}$$

In this case W refers to the length of the input, K to the width of the kernel, S to the stride and P to the amount of padding applied [43].

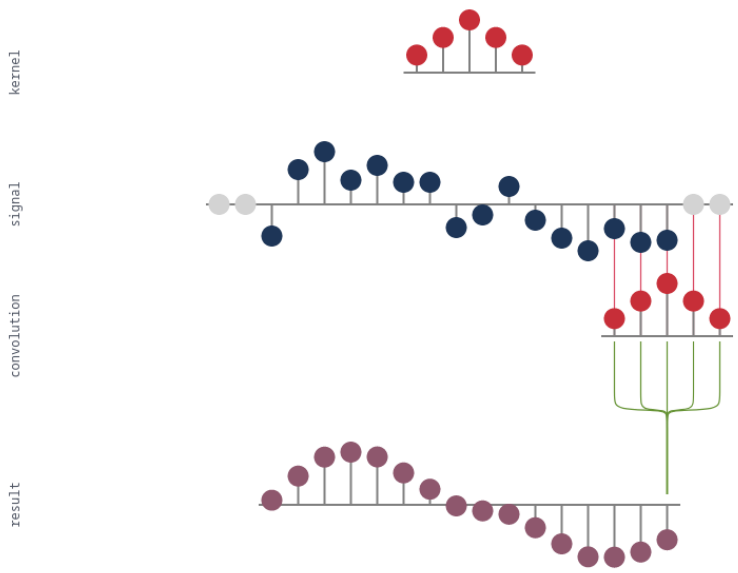


Figure 2.7: Convolution applied to a time series. The kernel has size five, padding is two and stride is 1. The kernel can be seen at the top and the result at the bottom. [1]

As mentioned earlier, the first part of most CNNs consist of an alternation of convolutional layers and pooling operations. These pooling operations are applied on the output feature maps and perform a down-sampling operation to reduce the data dimension of the input. This dimension reduction leads to a smaller number of parameters for the network to learn and therefore the amount of computation needed. Another effect are more abstract feature maps and a higher receptive field size of the following convolutional layers. The two most common pooling techniques are Max-Pooling and Mean-Pooling. While Mean-Pooling is a linear operation that can be seen as a smoothing operation, Max-Pooling is non-linear and can be seen as highlighting features that stand out from the rest [43].

In case of FCNs, the final output is the result of the last convolutional block plus any optional pooling or batch norm operations. However, if for example a classification should be performed, the last step in the chain of a convolutional NN is usually a fully-connected layer. It consists of a large number of neurons that are connected to all the output feature maps of the previous convolutional layer or pooling layer depending on the architecture. The fully-connected layer can be imagined like a basic multi-layer perceptron that takes a vector as an input and generates an output vector. In

[1] Plot was generated using a visualization tool by antoinebrl [2]

classification, the output vector has the same length as the amount of our classification classes if we have more than two classes. Usually when performing classification a softmax [7] function is applied to the output vector to generate a vector of classification probabilities for each class [22].

2.2.2 Training

The training of a deep NN and therefore also the training of CNNs is a complicated process with a lot of parts that need to be understood to succeed. As the training consists of several steps, each one of them will be explained in detail first and assembled to a bigger picture afterwards. If not stated otherwise, the following parts are derived from the writings of Szeliski [43].

Before starting to optimize the weights in the network, they first have to be initialized. While earlier approaches that used used small random weights had issues like progressively smaller activations in deeper layers, current approaches consider the number of incoming connections into a layer, the so called fan-in, to maintain a comparable variance in the activations of successive layers. A popular method used in combination with non-linear activation functions like ReLU is the He Initialization [18]. The authors state that the variance in the initial weights V_l for layer l should be set to

$$V_l = \frac{2}{n_l}$$

with n_l being the number of incoming activations for the layer.

To optimize the weights in a NN, we need some kind of objective function to maximize or minimize. For deep learning these are usually loss functions like cross-entropy. During training the loss from this objective function is minimized. The function for cross-entropy loss for a multi-class classification problem can be written as:

$$E(w) = \sum_n E_n(w) = -\sum_n \log p_{n_{t_n}} \tag{2.13}$$

where w contains all parameters of the model, p_{nk} is the networks current estimate of the probability of class k for sample n and t_n is the integer value of the sample class. The formula is based upon a final softmax layer in the model architecture.
However, if the task of the network is regression, a L2 Loss is often used:

$$E(w) = \sum_n E_n(w) = -\sum_n ||y_n - t_n||^2 \tag{2.14}$$

where y_n is the network output for the time series n and t_n is the target value.

While loss functions might seem more useful in supervised settings with a truth label to calculate the loss on, they have also proven their worth in unsupervised settings. An older example mentioned by Szeliski [43] is the contrastive loss function which was introduced by Hadsell, Chopra, and

LeCun [16]. The idea is to cluster similar samples while at the same time increasing the distance between dissimilar samples. By using an indicator whether the two provided samples are similar, an embedding can be computed such that similar input pairs have similar embeddings. The formula to compute the contrastive loss can be described as:

$$E_{CL} = \sum_{(i,j)\in\mathcal{P}} t_{ij} \log L_S(d_{ij}) + (1 - t_{ij}) \log L_D(d_{ij}) \tag{2.15}$$

P is the set of all labeled input pairs, L_S and L_D are similar and dissimilar loss functions and $d_{ij} = ||v_i - v_j||$ is the pairwise distance between paired embeddings [16]. An example network that leverages contrastive loss is the siamese network [10], which is trained by computing the loss after feeding a sample though both networks then utilizing it to backpropagate the gradients through both networks at the same time. Another method that works similar is the usage of triplet loss as shown by Hoffer and Ailon [19]. Triplet loss calculates the similarity between one sample and two other samples, one from the same class and the other one from another class. The goal is to ensure that the distance between samples from different classes is greater than the distance between matching samples [43].

With losses introduced, the next concept is backpropagation. After the weights have been initialized, they need to be iteratively modified until the network has converged to a set of weights that delivers a desired performance on the dataset. Weights are updated using a concept called backpropagation. Here the derivatives of the chosen loss function E_n for a specific sample n are calculated using the weights w and the chain rule. The calculation starts at the outputs and traverses through the network from back to front until it reaches the inputs. A visualization of the process of backpropagating the derivatives of the loss function through the network can be seen in 2.8. A derivative for the previously introduced exemplary L2 loss is:

$$\frac{\partial E_n}{\partial y_{nk}} = y_{nk} - t_{nk} \tag{2.16}$$

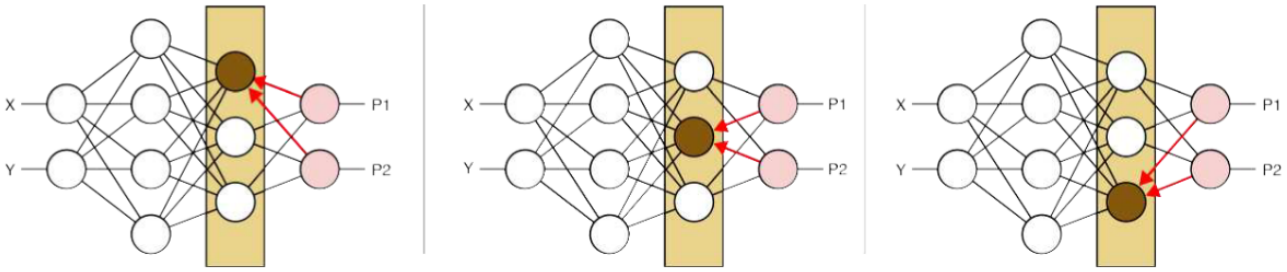


Figure 2.8: Backpropagation of the loss computed in the last layer to all nodes in the previous layer [43].

Assume that a units activation is defined as a weighted sum s_i by calculating the dot product between the weight vector w of that unit and its input activations:

$$s_i = w_i^T x_i + b_i = \sum_j w_{ij} x_{ij} + b_i \tag{2.17}$$

The weighted sum s_i from 2.17 is passed through an activation function h like ReLU to retrieve:

$$y_i = h(s_i)$$

Utilizing the chain rule the derivative of the loss can now be determined with respect to the bias, weights and input activations:

$$e_i = \frac{\partial E_n}{\partial s_i} = h'(s_i)\frac{\partial E_n}{\partial y_i}, \tag{2.18}$$

$$\frac{\partial E_n}{\partial w_{ij}} = x_{ij}\frac{\partial E_n}{\partial s_i} = x_{ij} e_i, \tag{2.19}$$

$$\frac{\partial E_n}{\partial b_i} = \frac{\partial E_n}{\partial s_i} = e_i, \tag{2.20}$$

$$\frac{\partial E_n}{\partial x_{ij}} = w_{ij}\frac{\partial E_n}{\partial s_i} = w_{ij} e_i \tag{2.21}$$

$e_i = \frac{\partial E_n}{\partial s_i}$ will be propagated through the network and is therefore defined as the error. To compute the error of a specific unit, a weighted sum of the errors coming from the units that it passes the output to is calculated. Afterwards it is multiplied by the derivative of the used activation function $h'(s_i)$. Subsequently, the loss derivative and backpropagation error are computed using equation 2.22 and 2.23.

$$\frac{\partial E_n}{\partial y_i} = \sum_{k>i} \frac{\partial E_n}{\partial x_{ki}} = \sum_{k>1} w_{ki} e_k \tag{2.22}$$

$$e_i = h'(s_i)\frac{\partial E_n}{\partial y_i} = h'(s_i)\sum_{k>i} w_{ki} e_k \tag{2.23}$$

While modern NNs like CNNs consist of a lot of other computational elements like convolutions, the previously described error propagation and derivatives are mostly unchanged.

The final step in the whole process of training a NN is the training algorithm itself, which uses all the previously described concepts to modify the networks parameters by using the computed gradients from 2.23 to improve our networks predictions. Modern implementations are based on the fundamental idea of stochastic gradient descent (SGD), which evaluates a single training sample n and computes the derivative of the loss $E_n(w)$ as seen in 2.22. This gradient is used as the direction for our optimization step.
However estimating the direction from just one sample is rather problematic, as that estimate would

be noisy to a large degree. To counter this problem, the data is randomly assigned into subsets, so-called minibatches. Afterwards the losses and gradients are summed over a single minibatch $\mathcal{B}$:

$$E_{\mathcal{B}}(w) = \sum_{n \in \mathcal{B}} E_n(w)$$

The gradient $g = \nabla_w E_{\mathcal{B}}$ is then used to update the weights by taking a step in the gradient direction:

$$w \leftarrow w - \alpha g \tag{2.24}$$

The parameter α that controls the magnitude of gradient update performed is called the learning rate. It is one of many parameters in deep learning that needs to be finely tuned to enable good learning progress while avoiding overshooting and exploding gradients [43].

There are various improvements that can be applied to SGD. Momentum is a concept that uses a decaying running average of the gradients to mitigate the stalling of gradient descent when reaching a flat spot. By using rather large values for $\rho \in [0.9, 0.99]$ the algorithm averages the gradients over multiple batches:

$$v_{t+1} = \rho v_t + g_t$$
$$w_{t+1} = w_t - \alpha_t v_t$$

Adam is one of the most popular optimizers for deep networks as it combines a lot of other improvement approaches into a single framework [43, 25]. However, setting the right hyperparameters so that the network achieves good performance within a reasonable training time is itself an open research area. This statement holds also for most of modern state-of-the-art implements where small changes in parameters can lead to significantly different results [43].

2.2.3 Regularization

Training a NN for too many epochs might result in an overfitting to the training data which leads to poor generalization performance on unseen data. To prevent this, regularization techniques exist. Data augmentation is understood as a regularization technique and will be described in detail in the next chapter.

One example that targets the loss function is the L2 norm. The L2 norm adds a penalty term to the loss function:

$$E_{\mathcal{B}}(w) = \frac{1}{n} \sum_{n \in \mathcal{B}} E_n(w) + \lambda ||w||_2^2$$

where λ is the regularization parameter. The L2 norm is also called ridge regression or weight decay and the main idea is that it forces the loss function to rather focus on large weights to reduce the loss function and therefore shrinks all weights, but especially large ones. It discourages complex

representations and increases stability by shrinking large weights [28].

Dropout is another regularization technique specially invented for NNs. During training with dropout a specific fraction p of nodes in the network are removed from the inference process (excluding the input and output layer). This process injects noise into the training process and prevents the networks nodes from over-specializing, therefore preventing overfitting and improving generalization [43, 42].

The inclusion of batch normalization [21] helps the network to converge more quickly. The prevention of internal covariate shift across one mini-batch training of time series is achieved by performing the batch normalization operation over each channel in a time series. The basic idea is to re-scale the activations at a given unit so that they have unit variance and zero mean. The computation for all samples n in a specific minibatch $\mathcal{B}$ is as follows:

$$\mu_i = \frac{1}{|\mathcal{B}|} \sum_{n \in \mathcal{B}} s_i^{(n)},$$
$$\sigma_i^2 = \frac{1}{|\mathcal{B}|} \sum_{n \in \mathcal{B}} (s_i^{(n)} - \mu_i)^2,$$
$$\hat{s}_i^{(n)} = \frac{s_i^{(n)} - \mu_i}{\sqrt{\sigma_i^2 + \epsilon}}$$

$\hat{s}_i^{(n)}$ is the normalized sum over a specific batch with zero mean and unit variance. Defining the output of the batch normalization as

$$yi = \gamma_i \hat{s}_i + \beta_i \tag{2.25}$$

by adding two additional trainable parameters γ and β allows the model to learn the optimal scaling and shift for the normalized activations. These parameters are trained just like regular weights. However, to prevent an immense amount of parameters in convolutional layers by computing the normalization for each time point, batch normalization is usually implemented by computing the statistics over all points with the same convolution kernel and then add a single gain and bias parameter to that kernel [22, 43].

2.3 Data Augmentation

It has been shown that an improvement of the model performance and ability to generalize can be achieved by increasing the amount of data [44], arguably, even more than tuning the model architecture [17]. The generalizability of a model refers to the performance difference of a model when evaluated on its training data versus data it has never seen before. Therefore, overfitting a model to its training data leads to poor generalizability.
However, the acquisition of large amounts of data can be a difficulty for many time series datasets [23].

One of the reasons is the cumbersome labeling process for time series. A good example to further convey this problem is the UCR Time Series Archive, which is one of the largest collection of datasets for time series. However out of the 128 only twelve include more than a thousand training patterns [12].

To address this challenge, a viable solution involves the generation of synthethic data samples, also known as data augmentation. In contrast to other regularization techniqes like batch normalization [21] or dropout [42], which primarily influence the training process of the network, data augmentations directly impact the training dataset. This implies that when utilizing data augmentations, there is an underlying assumption: the introduction of augmented data enriches the information available for training. This perspective is supported by research in the field [23, 40].

2.3.1 Types of Data Augmentations

Data augmentations are a common technique in computer vision tasks, particularly in image classification using NNs. AlexNet [26], one CNN mentioned as an important example in the previous chapter, leveraged data augmentations like cropping, mirroring and color augmentation. The data augmentations employed for AlexNet are (like most techniques used for images) based on random transformations of the training data [23]. However, there are various categories of data augmentations that can be applied to time series.
Several data augmentations are visualized in figure 2.9. The original sample is represented by a dashed blue line in all augmentation plots, with the augmented samples overlaid to emphasize the variations introduced by the depicted augmentation techniques.

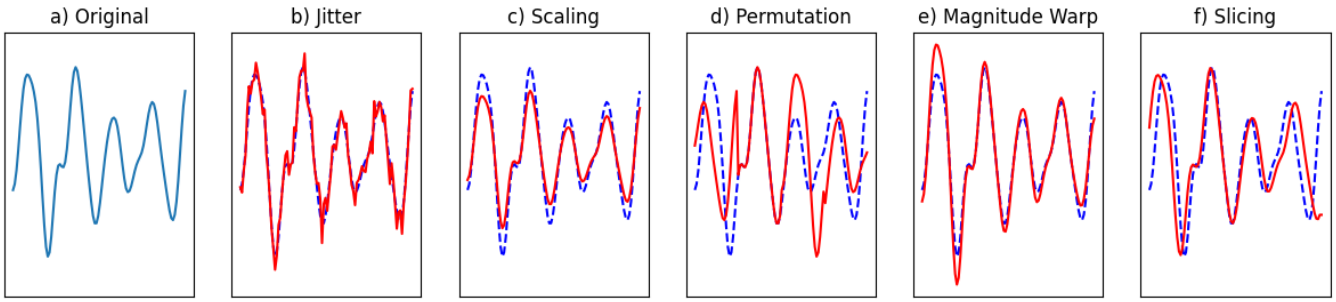


Figure 2.9: Examples of randomly generated data augmentations
a) Original sample waveform b)-f) Augmented samples overlaid on the original time series to illustrate the variations introduced by different augmentation techniques

Jittering, considered one of the simplest, yet effective data augmentations, is based on random transformations and can be applied to time series data [23]. It involves sampling $\epsilon_1, \ldots, \epsilon_T$ from a normal distribution with a mean of zero and a certain parameter σ. The term ϵ denotes Gaussian noise and can be expressed as $\epsilon \sim \mathcal{N}(0, \sigma^2)$. The entire jittering process can be viewed as adding

noise to inputs, a technique highlighted by An [1] as a means of enhancing the generalization of NNs. The sampled noise is added to each step:

$$X' = x_1 + \epsilon_1, \ldots, x_T + \epsilon_T \tag{2.26}$$

An illustrative application of jitter can be observed in figure 2.9. While the sample retains its original identity, the introduction of noise disrupts the visual continuity or introduces visual irregularities.

Scaling is another data augmentation technique based on random transformations that influences the magnitude of a time series. It alters the intensity of a time series by a random value and can be formulated as:

$$X' = \alpha x_1, \ldots, \alpha x_t, \ldots, \alpha x_T \tag{2.27}$$

Magnitude warping performs transformations in the magnitude domain by warping the magnitude of a time series using a smoothed curve. The augmentation can be expressed as:

$$X' = \alpha_1 x_1, \ldots, \alpha_t x_t, \ldots, \alpha_T x_T \tag{2.28}$$

Here, $\alpha_1, \ldots, \alpha_t, \ldots, \alpha_T$ are generated by interpolating a cubic spline $S(u)$ with knots $u = u_1, \ldots, u_i, \ldots, u_I$. A total of I knots are selected from $\mathcal{N}(1, \sigma^2)$. These transformations introduce small added fluctuations in the data by randomly increasing or decreasing regions in the time series [23].

While the preceding examples have been transformations in the magnitude domain, emphasizing values within the time series, slicing focuses on transformations along the time axis. Slicing is similar to cropping in the image space and is defined by the equation:

$$X' = x_\varphi, \ldots, x_t, \ldots, x_{W+\varphi} \tag{2.29}$$

Here, W represents the size of the window, and φ is a random integer such that $1 \leq \varphi \leq T - W$. The fundamental idea behind slicing is to extract a random part with a size of W from the time series. Given that the length of the extracted sub-sample is smaller than the original sample, the time series undergoes linear interpolation to match the length of the original data samples [23].

An additional augmentation in the time domain is permutation, involving the rearrangement of extracted parts of a time series to generate a new pattern. Since it alters the ordering of observations within a signal, permutation does not preserve time dependencies. This is clearly illustrated in figure 2.9.

Introducing transformations in the frequency domain is another way of augmenting random transformation-based. An example of this is frequency warping, where implementations like Vocal Tract Length Perturbation (VTLP) [24] map the frequency of the time series to a new frequency using warping.

While usually being the most prominent ones, random transformation-based data augmentations are not the only type of data augmentations. Another category involves augmentations that synthesize new time series by leveraging inherent properties of the data. While random transformations may inadvertently alter the distribution, according to Iwana and Uchida [23], one advantage of synthetization methods is their attempt to preserve the original distribution of time series within the dataset. These augmentations can be further classified into pattern mixing, generative models and pattern decomposition methods.

While the basic idea behind pattern mixing is the combination of two or more patterns to generate new data, there are multiple ways to achieve this. Patterns can either be mixed using the magnitude, time, frequency domain or all of them at the same time. One example for mixing in the magnitude domain is Synthetic Minority Over-sampling Technique (SMOTE) [9]. The authors address imbalanced datasets by interpolating samples from underrepresented classes. The new sample can be calculated as

$$X' = X + \lambda |X - X_{NN}| \tag{2.30}$$

with λ being a random value in the range of $\{0, 1\}$, X being a sample from the underrepresented class and X_{NN} being a random sample from X's k-nearest neighbors [23].

Instead of utilizing the raw time series values, generative models can sample time series from feature distributions. These models can be either statistical or neural network-based. An example of the latter category includes generative adversarial networkss (GANs) [15]. GANs employ adversarial training, optimizing a pair of NNs simultaneously: a generator and a discriminator. The discriminator network assesses the quality of the synthesized output, contributing to the training of the generator. After training the network using the dataset, a random vector is sampled and fed into the generator to generate a new time series sample. Various GANs exist for data augmentation, incorporating approaches based on CNNs, recurrent neural networks (RNNs) or simply fully-connected networks [23].

The final approach is the synthetization of samples using time series decomposition. As already mentioned in section 2.1.1, decomposition methods in general decompose time series signals by extracting features or underlying patterns like trend, seasonality and a remainder. These extracted features can then be used alone or combined to generate new data for augmentation purposes. Bergmeir, Hyndman, and Benítez [6] leverage STL for decomposition to afterwards bootstrap the remainder using a moving block bootstrap in combination with a Box-Cox transformation. The new signal is subsequently generated by using the bootstrapped remainder [23].

2.4 Visualization Techniques

The process in training highly performant, deep NNs has been immense. For example, AlexNet's [26] training process highly benefited from significantly lower training time utilizing parallelization on

GPUs, leveraging new techniques like dropout [42] and ReLU [33], and the usage of data augmentations like cropping and flipping. However, while the knowledge about creating high-performant deep learning architectures has expanded, according to Yosinski et al. [48] our understanding of how those large NNs function has not improved by nearly the same factor. Especially large NNs are described as black boxes as the huge amount of interacting, non-linear parts make it hard to comprehend how they exactly operate. While understanding the process inside of deep NNs might be interesting on its own, it is also a major opportunity to further improve the models by leveraging the intuitions provided by comprehension through visualization. A notable example is the work by Zeiler and Fergus [49], where they identified issues in their first and second convolutional layer using their proposed visualization techniques. Addressing these problems enabled them to surpass the state-of-the-art architecture at that time.

As the research regarding image visualizations is a lot more advanced than it is for time series, the next section will focus on visualizations techniques for CNNs for images. While most of these are also applicable to 1D convolutions with time series, the section afterwards will focus on techniques specifically designed for time series.

2.4.1 CNN Visualization Techniques

According to Qin et al. [37], the demonstration of the internal features learned by CNNs can improve the interpretability of the working mechanisms of these networks. The authors state that especially visualization helps greatly to interpret those features as it utilizes the human visual cortex system as a reference.

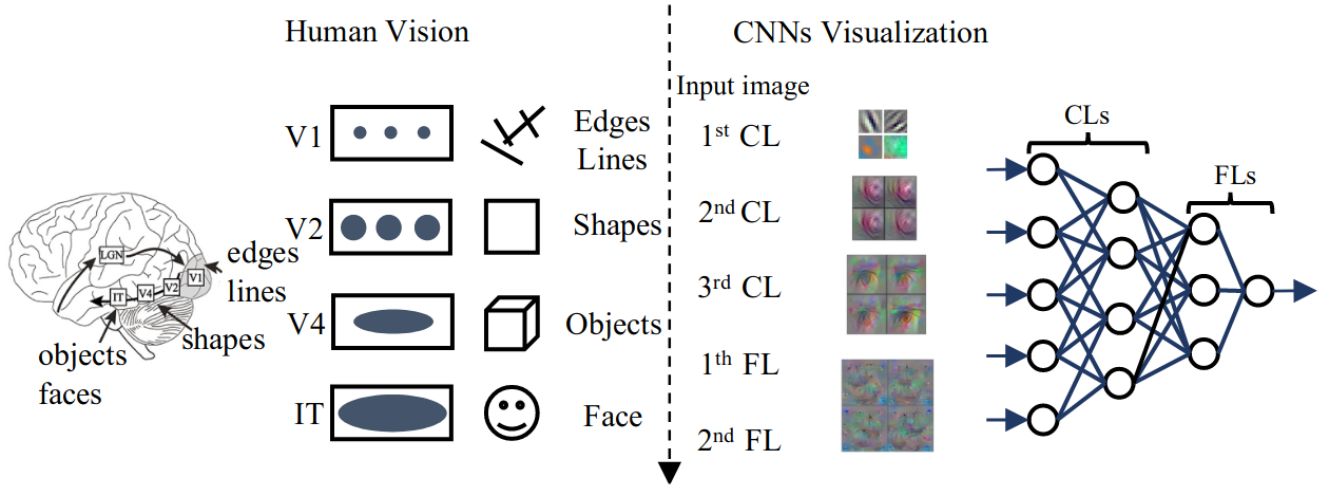


Figure 2.10: Visualized comparison of the human vision system and features extracted by a CNN. On the left, it can be observed how the visual cortex system processes visual information in a feed-forward approach through multiple visual neuron areas. On the right, the extraction of increasingly more complex features from layer to layer can be seen [37].

Figure 2.10 depicts the process of human vision and a visualization of how CNNs extract features. While neurons with lower visual neuron area like V1 are more sensitive to basic features such as lines, the neurons in higher visual neuron areas with larger receptive fields are strongly triggered by more complex features like shapes and objects. Something similar can be observed in CNNs, where the first convolutional layer typically extracts small features like edges, while deeper layers mostly respond to shapes and parts of objects.

A method that provides further understanding of intermediate feature layers and the operation of the classifier is the deconvolution method [49]. It is a technique to reveal the inputs that activate specific features maps at any layer using a so-called multi-layered deconvolution network. The deconvolution network is used to project the features activations back to the input pixel space visualizing which input pattern caused the given activation in those feature maps. It can be conceptualized as a convolutional network with the same components but in reverse. The specific architecture can be seen in figure 2.11 where the deconvolution network is attached to the CNN establishing a path back to the image space.

As a first step an input image is processed by the CNN and all feature maps are saved in the process. Examining a specific CNN activation consists of setting the other activations of the layer to zero and passing the feature map as input to the corresponding layer of the attached deconvolution network. Afterwards an alternation of unpooling, rectifying and filtering is used until input image space is reached. As the max pooling operation of the CNN is not reversible, the inverse is approximated by recording the maxima locations in a set of switch variables which can also be seen in figure 2.11. The convolutional filters used are flipped vertically and horizontally representing the inverted convolution process.

An example application of deconvolution can be seen in figure 2.12. Five convolutional layers are shown, where in each layer two randomly selected neurons' visiualized patterns are compared to the corresponding local settings in the original image. It can be seen that each individual neuron extracts features in a more local manner, where different neurons in one layer are responsible for different patterns like mouth, eyes and ears [37]. While lower layers capture less abstract features such as edges, corners and small parts, it can also be observed that higher layers on the other hand are more class-specific and show almost entire objects. Simonyan, Vedaldi, and Zisserman [41] performed similar work by demonstrating the projection back from the fully connected layers of the network instead of the convolutional layers.

The visualization of inputs that excite a certain neuron in each layer is another approach for trying to further understand CNNs, called activation maximization (AM). In order to synthesize an input pattern that produces maximal activation of a neuron, each pixel of the input is changed iteratively using gradient ascent. While Simonyan, Vedaldi, and Zisserman [41] used AM to maximize activations in the last convolutional layer, Yosinski et al. [48] leveraged the algorithm on a large scale and applied it to all units in all layers of a CNN. With a lot of other work being done in this

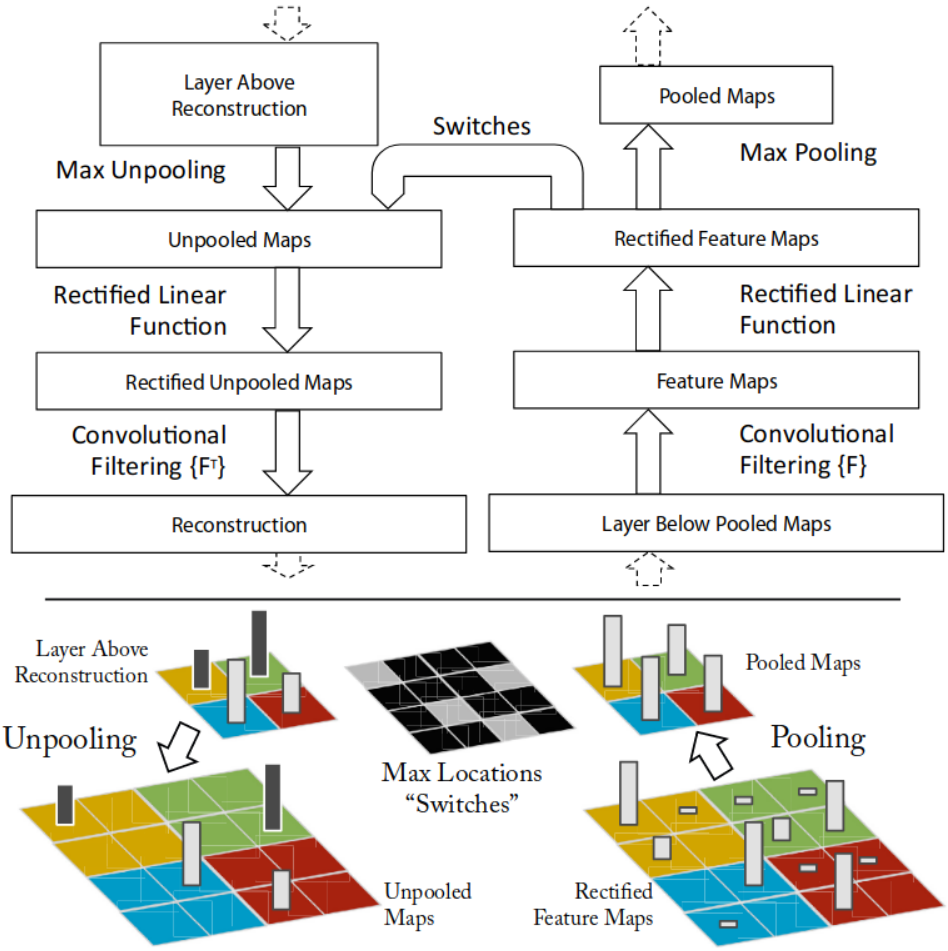


Figure 2.11: Deconvolution process. A deconvolution network (left) is attached to a convolutional neural network on the right. The unpooling operation utilizing switches to record the local maxima in each region during pooling is visualized at the bootm. The deconvolution process reconstructs an approximate version of the features of the CNN from a certain layer. [49]

regard, AM has proven its great capability to interpret the interests of neurons and identify features learned by CNN. Examples for images generated by AM can be seen in figure 2.13.

Six hidden layers are shown with several neurons in each layer randomly selected. It can be observed that lower layers react to basic features like edges and higher layers react to more complex features like faces in the first picture of the fourth layer. However, even after applying several regularizations some visualized patterns are not interpretable [37].

The synthesized image, which maximizes the activation of a target neuron can be formulated as:

$$X^* = \underset{X}{\operatorname{argmax}} f^{(l,u)}(X) \tag{2.31}$$

where $f^{(l,u)}$ computes a representative value of the feature map of a specific unit u in layer l in

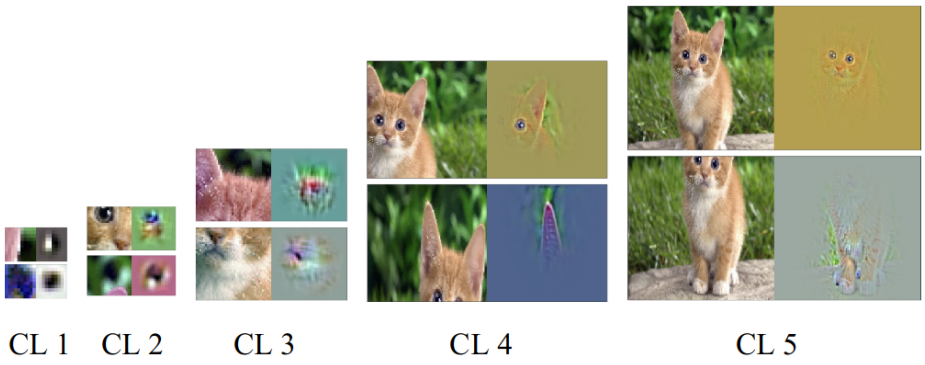


Figure 2.12: Deconvolution example with CaffeNet taken from Qin et al. [37]. Two randomly selected visualized patterns are shown for five convolutional layers. The corresponding local sections in the images are shown next to the visualized pattern.

regards to equation 2.17.
The first step is to create an initial image and assign it $X = X_0$. While there are various methods to choose a starting image, random noise seems to perform well. Afterwards the gradients with respect to the noise image $\frac{\partial f^{(l,u)(X)}}{\partial X}$ are calculated using backpropagation with fixed weights. The update step applied each iteration now consists of changing the input image in the direction of the gradient with the gradient ascent step size η:

$$X \leftarrow X + \eta \frac{\partial f^{(l,u)}(X)}{\partial X} \tag{2.32}$$

The gradient ascent in this case is adapted from the gradient descent application in equation 2.24 with the goal to maximize the value of $f^{(l,u)}(X)$. The procedure in equation 2.32 is typically iterated for a designated number of steps before concluding with the identification of a specific image X^* that maximally activates the targeted neuron and therefore fulfills equation 2.31 [37].

However, applying the method to deep convolutional layers often produce results that seem unrealistic and uninterpretable. Therefore regularizations $R(X)$ can be applied to bias generated images toward more visually interpretable examples:

$$X^* = \underset{X}{\text{argmax}}(f^{(l,u)}(X) - R(X)) \tag{2.33}$$

An exemplary regularization technique is Gaussian blur to combat high frequencies in generated images. While these images with high frequencies might cause high activations, they are neither realistic nor interpretable and can therefore be penalized. Therefore, applying a Gaussian blur step can help with generating more meaningful images [48].
It should be noted that other approaches for AM such as the utilization of GAN exist [35].

A method that indicates discriminative image regions used by the CNNs to identify specific categories is class activation maps (CAMs). Zhou et al. [51] show that their approach is able to localize

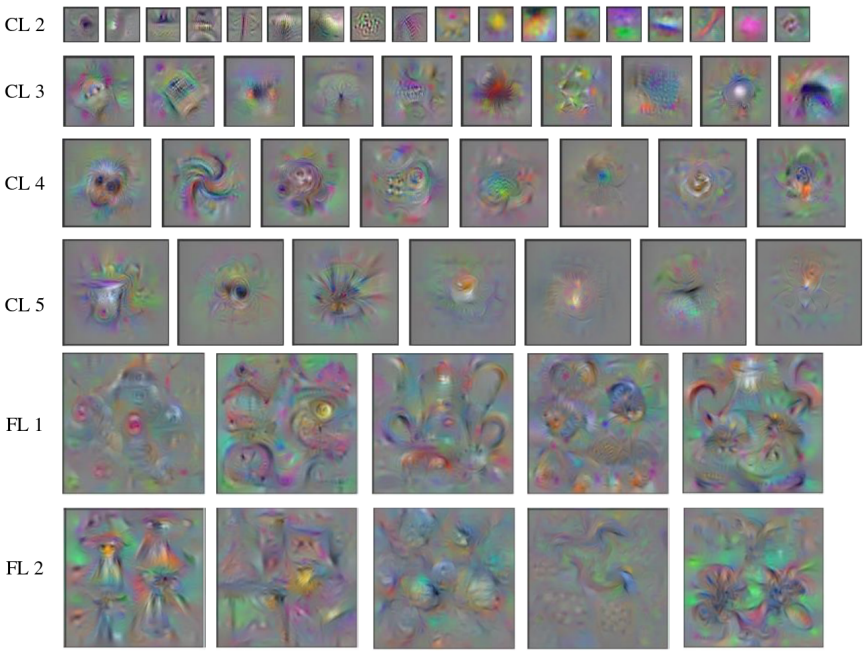


Figure 2.13: AM visualizations of CaffeNet. Visualizations are included for all convolutional and fully connected layers. For each layer several neurons are randomly selected. The complexity of the extracted features seems to increase with each layer. [37]

discriminative image regions by leveraging a network that consists mainly of convolutional layers with a global average pooling layer just before the CNN. CAMs are created by projecting back the weights of the output layer on to the convolutional feature maps, thereby enabling the basic understanding of discrimination used by CNNs for their tasks. There are a lot of approaches based on this concept, one popular example being Grad-CAM by Selvaraju et al. [38]. A few examples images for CAMs can be seen in figure 2.14.

2.4.2 Time Series Techniques

The groundwork laid by Zhou et al. [51] in the development of CAMs has been further applied to time series analysis by subsequent researchers. Leveraging a global averaging pooling layer enables the CNN to find the contributing regions in the raw time series for specific classes without a lot of extra computation. Therefore, it provides an option to find explanations on how CNNs perform a prediction [46]. Should the output length of the last convolution layer not match the length of the input time series due to convolutions or pooling operations, the CAM can be upscaled to match the input length. An example for the visualization of CAMs on time series can be seen in figure 2.15. According to Ismail Fawaz et al. [22], it can be seen that the model neglects the plateau as a non-discriminative region of the time series when taking the classification decision. An evidence for CNNs ability to learn time-invariant features is that the model is able to localize a specific discriminative shape regardless of its temporal location.

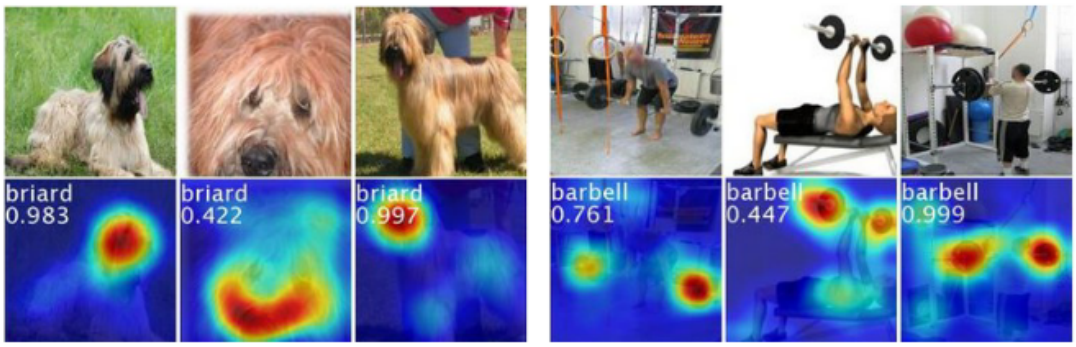


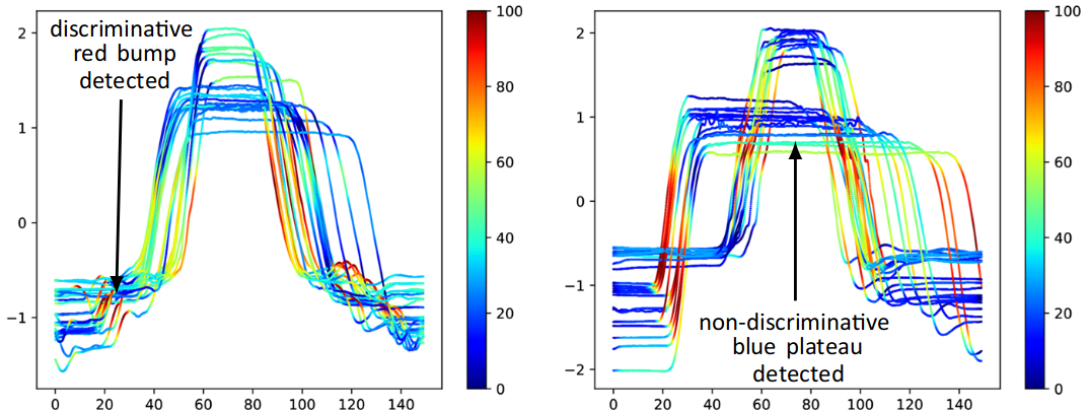
Figure 2.14: Examplary Class Activation Maps for two different classes. The row above shows the original image, while the row on the bottom highlights the discriminative image regions used for image classification for those networks. The head of the animal and the plates of a barbell seem to play a major factor in the models prediction [51].

While methods like CAM are able to indicate important regions that the model uses when performing a classification, they are not capable of revealing the underlying features extracted by the NN. However, applying AM, a visualization technique commonly used in computer vision capable of visualizing extracted features, to CNNs for time series data poses a challenge. According to Yoshimura, Maekawa, and Hara [47], the conventional AM methods applied to specific time series, particularly acceleration-based data, result in meaningless and noisy signals. This issue arises due to difficulties in directly utilizing the values of acceleration signals during the regularizing of AM. Therefore, the authors propose new regularization techniques for AM on time series data by leveraging activation values in feature maps to facilitate the research and usage of CNNs for time series when attempting to understand and optimize these networks. As those regularization techniques play an important role in this bachelor thesis, the following part will explain their regularization techniques in detail.

With X defined according to equation 2.1, having a length of T and number of channels C, the input to the network is of size $(C \times T)$.
$X^{(l,u)}$ is used to denote the output of unit u in layer l of the network, resulting in a $(C' \times T)$ matrix, where C' depends on the amount of kernels in the l-th convolutional layer. The input X^* that maximizes the activation of unit u in layer l can be computed similarly to images, as seen in equation 2.31. However, the generated X^* contain high-frequency noise, rendering them unrecognizable to humans [47, 34]. Introducing a regularization term $R(X)$, as shown in formula 2.33 can be beneficial to reduce those high-frequency components. To solve equation 2.33, gradient ascent is employed to maximize the value of $f^{(l,u)}(X) - wR(X)$. The process starts by randomly initializing X and then iteratively updaing it using gradient ascent:

$$X \leftarrow X + \eta \frac{\partial}{\partial X}(f^{(l,u)}(X) - wR(X)) \tag{2.34}$$

Here, η represents the learning rate as in equation 2.24.



(a) FCN on GunPoint: Class-1 **(b)** FCN on GunPoint: Class-2

Figure 2.15: Class Activation Maps for time series on UCR GunPoint Dataset taken from Ismail Fawaz et al. [22]. The contribution of each time series region is highlighted for both classes in the UCR GunPoint dataset. Red corresponds to a higher contribution, while blue represents almost no contribution to the correct class identification.

Given that $x_{c,t} \in X$ represents the observation at a specific channel c and time t, as defined in equation 2.1, the first of two regularizations, also employed by Yosinski et al. [48], is the Lp norm, which can be defined as:

$$R_{Lp}(X) = \left(\sum_{c}^{C-1} \sum_{t}^{T-1} (x_{c,t})^p \right)^{\frac{1}{p}} \tag{2.35}$$

Total variation on the hand, another regularization term used by the authors, can be computed with:

$$R_{TV}(X) = \left(\sum_{c}^{C-1} \sum_{t}^{T-2} |x_{c,t} - x_{c,t+1}| \right) \tag{2.36}$$

While AM might produce extreme activations, simply restricting the value range of the generated data by clipping will not produce meaningful results as time series data generally does not exhibit an upper or lower limit and different units in the network have different sensitivities in terms of input signal amplitudes. The first regularization term proposed by Yoshimura, Maekawa, and Hara [47] leverages the distribution of activation values for each unit when feeding all the training data through the network. This enables the definition of a sufficient activation value range for each node based on its own distribution. The threshold is defined as $th_{eap}^{(l,u)}$ and used in their proposed regularization term called extreme activation penalty (EAP):

$$R_{EAP}(X^{(l,u)}) = \text{mean}(\{a_{c,t}^{(l,u)} - th_{eap}^{(l,u)} | a_{c,t}^{(l,u)} \in X^{(l,u)}; a_{c,t}^{(l,u)} > th_{eap}^{(l,u)}\}) \tag{2.37}$$

where $a_{c,t}^{(l,u)} \in X^{(l,u)}$ represents the activation value at channel c and time t. The threshold value is determined using the two-sigma rule, assuming a normal distribution for activation values that effectively encompasses up to the 95th percentile of the activation values in the training data. The

penalty term is then calculated based on the average of exceeded values [47].

The second regularization method is called activation clipping and is supposed to supress extreme activation values that are challenging for EAP. As other regularization terms use the activations representative value $f^{(l,u)}$, for example a mean function, it is is difficult to target a specific individual extreme activation value in $X^{(l,u)}$. Therefore activation clipping is applied to each individual value in the feature map before computing the representative value. To apply clipping with the previously used threshold $th_{eap}^{(l,u)}$ and thereby mitigating explosions of extreme activation values, the following formula can be used [47]:

$$clipping(a_{s,t}^{(l,u)}) = \begin{cases} th_{eap}^{(l,u)} & (a_{c,t}^{(l,u)} > th_{eap}^{(l,u)}) \\ a_{c,t}^{(l,u)} & \text{(otherwise)} \end{cases}$$

Applying these regulations enabled the generation of time series that are more similar to the original training data. However, as exploring the generated signals for one unit at a time is not feasible with the large amount of hidden units in a NN, Yoshimura, Maekawa, and Hara [47] develop a so-called activation atlas similar to Carter et al. [8]. The activation atlas enables an easier analysis of extracted features representations and one example for the first layer of a CNN can be seen in figure 3.14.

2.5 Datasets

To offer deeper insights into both univariate and multivariate datasets, two datasets have been selected. The first dataset, called FordA, is sourced from the widely recognized UCR Time Series Classification Archive [12]. The second dataset is known as the DSADS dataset [3]. Subsequent subsections will provide detailed descriptions of each dataset.

2.5.1 UCR - FordA

FordA, an univariate dataset, consists of 500 measurements of engine noise. Based on this noise the task is to diagnose whether a certain symptom exists or not in an automotive subsystem. For this specific dataset, both the train and test data was collected in operating conditions with minimal noise contaminations. The training dataset consists of 3601 samples, whereas the test dataset contains 1320 samples. As the task is a boolean prediction, the number of classes is two. Two example signals for each class can be seen in figure 2.16 [12].

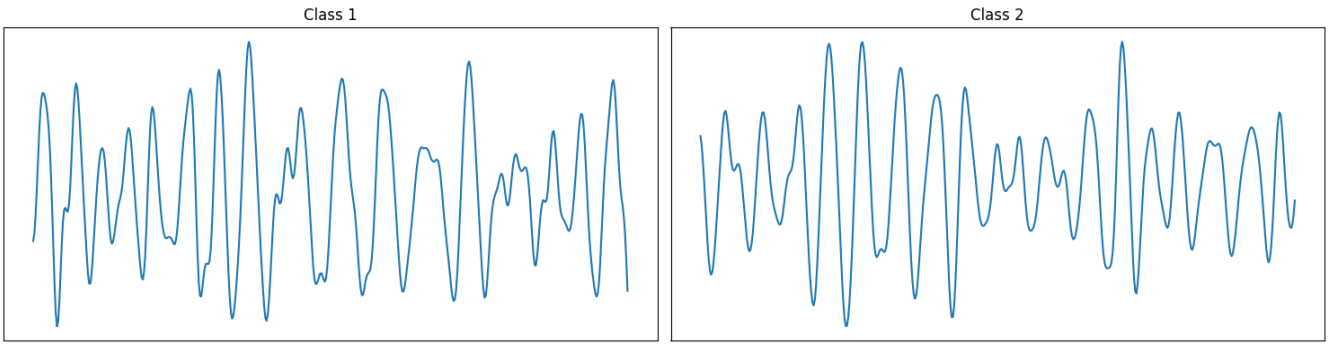


Figure 2.16: Exemplary line plots for random samples for each class of the FordA dataset.

2.5.2 Daily Sports and Activities

The DSADS multivariate dataset, established by Barshan and Altun [3], comprises motion sensor data representing 19 distinct daily and sports activities. The 19 different activities are sitting, standing, lying on back and on right side, ascending and descending stairs, standing in an elevator still, moving around in an elevator, walking in a parking lot, walking on a treadmill with a speed of 4 km/h (in flat and 15 deg inclined positions), running on a treadmill with a speed of 8 km/h, exercising on a stepper, exercising on a cross trainer, cycling on an exercise bike in horizontal and vertical positions, rowing, jumping, and playing basketball.

The dataset involves the participation of 8 different subjects (4 female, 4 male, between ages 20 and 30), each engaging in every activity for a duration of 5 minutes without specific instructions. The total signal duration for each subject is 5 minutes; however, the signals are segmented into 5-second intervals, resulting in a sample length of 125 measurements with a sampling frequency of 25 Hz. Therefore, each activity has a total number of 480 samples as there are 60 samples for each participant.

Every subject was outfitted with 5 sensor units placed on distinct parts of their body: torso, left and right arm, and left and right leg. Since each sensor unit records 9 different values (x, y, z accelerometer; x, y, z gyroscope; x, y, z magnetometer), the dataset is characterized as a multivariate dataset. For each sample, there are 45 time series, derived from the combination of 5 sensor units and their respective 9 sensor measurements [3].
Example visualizations for this dataset can be found in figure 2.1 (section 2.1).

2.6 Algorithms

2.6.1 t-SNE

t-SNE is a dimensionality reduction technique designed to visualize complex patterns in high-dimensional data. It aims to preserve pairwise similarities between data points by performing two main steps.
First high-dimensional similarities p_{ij} are computed using a Gaussian distribution:

$$p_{ij} = \frac{exp(-\frac{||x_i - x_j||^2}{2\sigma_i^2})}{\sum_{k \neg i} exp(-\frac{||x_i - x_k||^2}{2\sigma_i^2})}$$

where σ_i represents the variance of the Gaussian distribution centered around x_i.
The second distribution necessary is Student's t-distribution which is used to model similarities in lower-dimensional space:

$$q_{ij} = \frac{(1 + ||y_i - y_j||^2)^{-1}}{\sum_{k \neg i}(1 + ||y_k - {y_i}^2)^{-1}}$$

To find a lower-dimensional representation that maintains the similarities of both high and low dimensions, the Kullback-Leibler divergence between the two distribution is minimized:

$$C = \sum_i KL(P_i||Q_i) = \sum_i \sum_j p_{ij} \log(\frac{p_{ij}}{q_{ij}})$$

This minimization process is performed leveraging gradient ascent [30].

2.6.2 k-Means

k-Means is a clustering algorithm commonly utilized for unsupervised learning and data segmentation. In this algorithm, a dataset is partitioned into clusters based on the similarity of the data points. Each cluster is represented by a centroid, and these centroids' positions are iteratively updated to minimize the sum of squared distances between data points and their assigned centroids. This sum of squared distances is described by the following function:

$$J = \sum_{i=1}^{n} \sum_{j=i}^{k} ||x_i - c_j||^2$$

Here, x_i denotes a specific data point and c_j represents its assigned centroid.
After initializing centroids, each data point is assigned to its closest centroid. The position of centroids are then recalculated using the mean of all data points assigned to that cluster. Subsequently, data points are reassigned to their closest centroid again. These steps are repeated for a certain amount of steps or until a certain threshold is reached [31].

3 Methodology

In the following chapter, a comprehensive exploration of the research methods and techniques employed to extract results will be provided. This detailed discussion will include the methodology's design, quantitative approaches and any specific tools utilized in the research process. Figure 3.1 provides an illustration of the interaction between various components in the methodological approach for this thesis. TimeLens-Web, a Next.js web application, takes responsibility for interactively displaying all results to the user. To achieve this, it accesses FastAPI, a python library for building application programming interfaces (API), utilizing an API request. FastAPI manages the inference of PyTorch models, while also leveraging TimeLens to facilitate access to raw values necessary for visualizations. To generate those raw values, TimeLens also accesses the already trained PyTorch models.

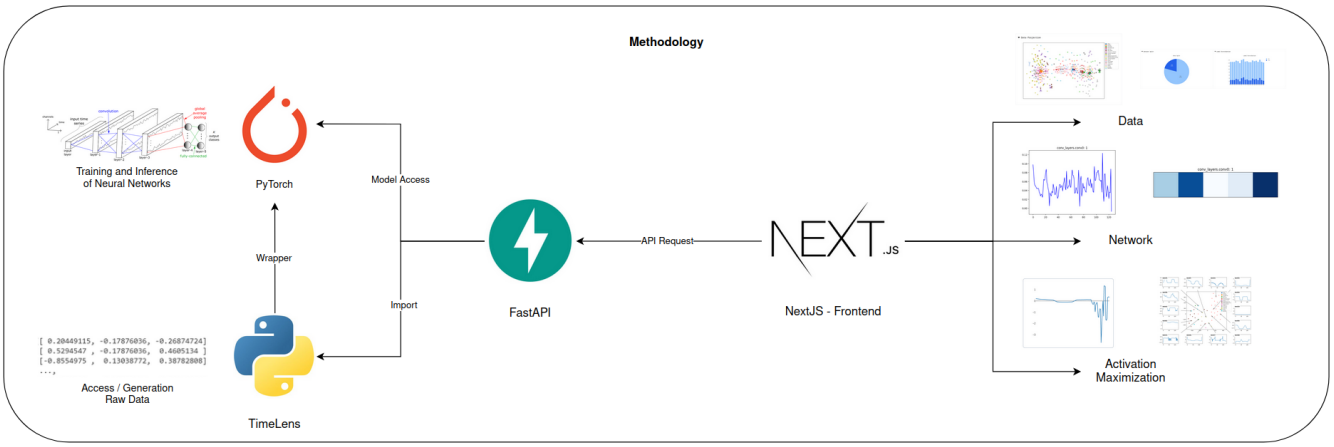


Figure 3.1: Visualization of all interactions between components for this thesis. On the left, deep learning models are trained by PyTorch and accessed by TimeLens. FastAPI provides an easy API access to both components. On the right, Next.js provides a web interface to interact with all visualizations generated by utilizing the API access to FastAPI.

3.1 TimeLens

TimeLens is a Python package developed in the course of this thesis to enhance the understanding of CNNs for time series. Serving as the foundation for other tools, TimeLens offers easy access to

raw network data, visualizations of kernels or feature maps and the evaluation and processing of optimized activation maximization.

The raw data extraction involves wrapping a *CNNRaw* class from TimeLens around any PyTorch CNN. This wrapper provides several methods for accessing kernel weights for a specific kernel or a whole layer, as well as extracting layers of feature maps or specific feature maps for a given time series input. The extraction of weights in the form of kernels involves retrieving a specific convolutional layer by its name and using PyTorch functions to access the actual weights. If necessary, further processing such as normalization is performed, and subsequently, a specific unit or the entire layer is selected.
On the other hand, to obtain feature maps, PyTorch hooks are registered on a specific convolutional layer. These hooks are executed whenever the forward function is called for the module containing the layer, saving the feature maps for later access. Therefore, to extract feature maps, hooks are set up and an input time series is subsequently passed through the network, allowing access to the feature maps.

The visualization component of the package utilizes the earlier extracted raw data to facilitate the network exploration through visual representations. It supports plotting of kernel data by normalizing the kernels and displaying them as heatmaps. Using an additional input time series, the same process is also applicable for feature maps. In figure 3.2, a normalized kernel heatmap, as generated by TimeLens, is presented. In this heatmap, where brighter and darker values indicate smaller and larger values, respectively. On the other hand, feature map visualizations are plotted with the time steps on the x-axis and the magnitude on the y-axis. Both of these visualizations offer a robust foundation for exploration.

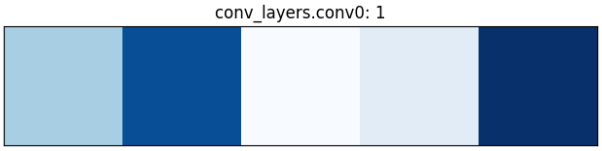


Figure 3.2: Exemplary Kernel Visualization of a CNN (first convolutional layer)

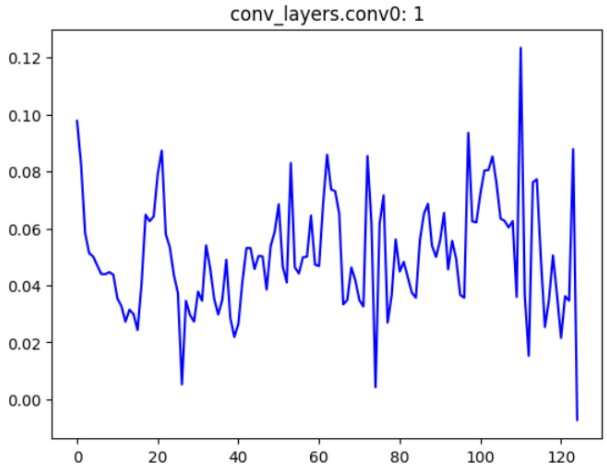


Figure 3.3: Exemplary Feature Map Visualization (first convolutional layer)

Another aspect covered by TimeLens is activation maximization (AM), which is largely based on the

work by Yoshimura, Maekawa, and Hara [47], as explained in detail earlier in section 2.4.2. The implementation details for this component will be discussed in section 4.1.
TimeLens supports various inputs for AM and incorporates penalization techniques to facilitate the generation of meaningful and interpretable maximum activations. Following the generation for the whole layer or a specific input, the AM process can be evaluated by assessing the similarity of generated AM signals to the original training data.

3.2 TimeLens-Web

Exploring visualizations using only Python and its function calls may be cumbersome. To address this, a web-based visualization tool called TimeLens-Web was developed. This tool enables an interactive approach to explore different models with their data augmentations in various ways.

The usage and flow of the application will be explained in the following: The initial option presented is the data augmentation that the model has been trained with. As illustrated in figure 3.4, an augmentation can be selected from a number of choices and is highlighted once chosen. Afterwards all subsequent visualizations will utilize the model trained with that specific data augmentation. The tool further separates the visualization options into three categories as seen in figure 3.5: Data, Network, Activation Maximization.

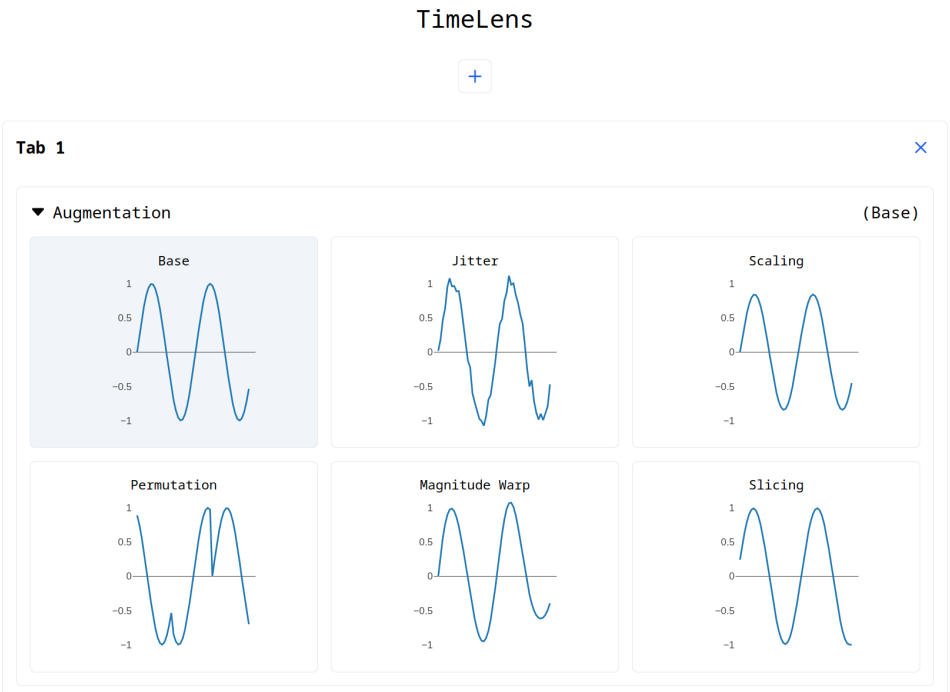


Figure 3.4: Selection of a data augmentation to apply in TimeLens-Web. The chosen data augmentation is highlighted in blue.

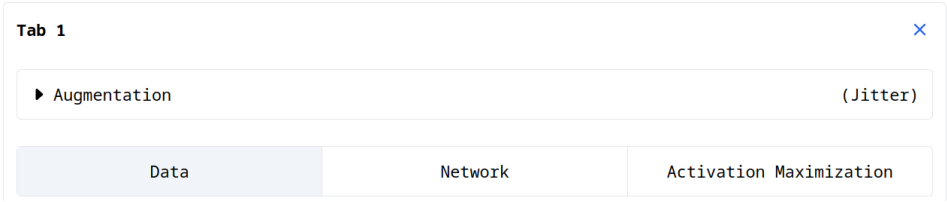


Figure 3.5: Selection of a feature tab in TimeLens-Web after selecting a data augmentation. There are three options: Data, Network and Activation Maximization.

Regarding the data tab, two visualizations supporting the process of finding any imbalances in the training dataset are shown in figure 3.6. The left visualization demonstrates the ratio of training to test data, while the right one illustrates the number of samples for each class. This enables a quick identification of insufficient samples for a class, potentially leading to poor prediction performance. Although these two plots are separated and displayed below each other in the web application, they have been adjusted to be visualized side by side for easier integration into this thesis.

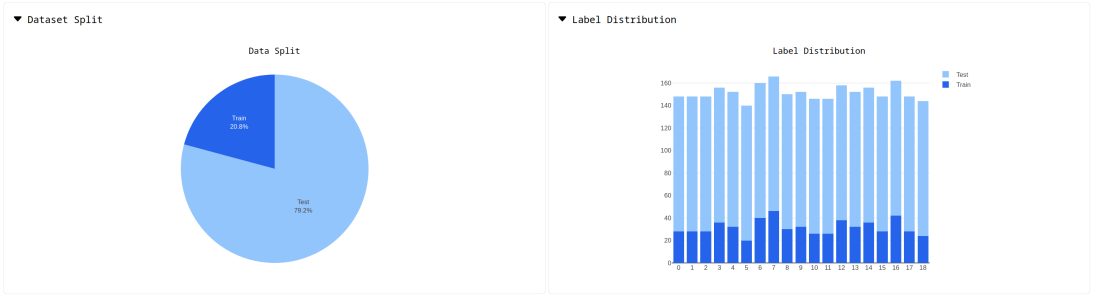


Figure 3.6: Visualization of a train/test split of the training dataset on the left and the distribution of labels for both the training and test dataset on the right. Both are accessed by opening up the corresponding collapsible section.

Data augmentations usually generate synthetic data that deviates from the original dataset. However, as the degree of deviation is often unknown, it is helpful to project both the original data and generated samples into 2D space using dimensionality reduction techniques such as t-SNE [30] for visualization purposes. To do so all original and generated signals used in the training process are reduced to two components, enabling them to be visualized in a scatter plot. Applying jitter to a subset of the DSADS dataset leads to the plot in figure 3.7. Augmented samples can be differentiated from original ones by their fill grade. While original samples are filled, the generated ones are unfilled. By observing the distances between a sample and its generated counterpart, a first sense of the deviation can be gained.

The second tab, called Network, allows the exploration of kernels and leraned features of a given CNN.
Kernels from a CNN are extracted utilizing the TimeLens package and can be viewed by accessing the kernel section of the web application. The visualization and options available are depicted in figure 3.8. Options include changing the plot style from a heatmap to a line plot, choosing which

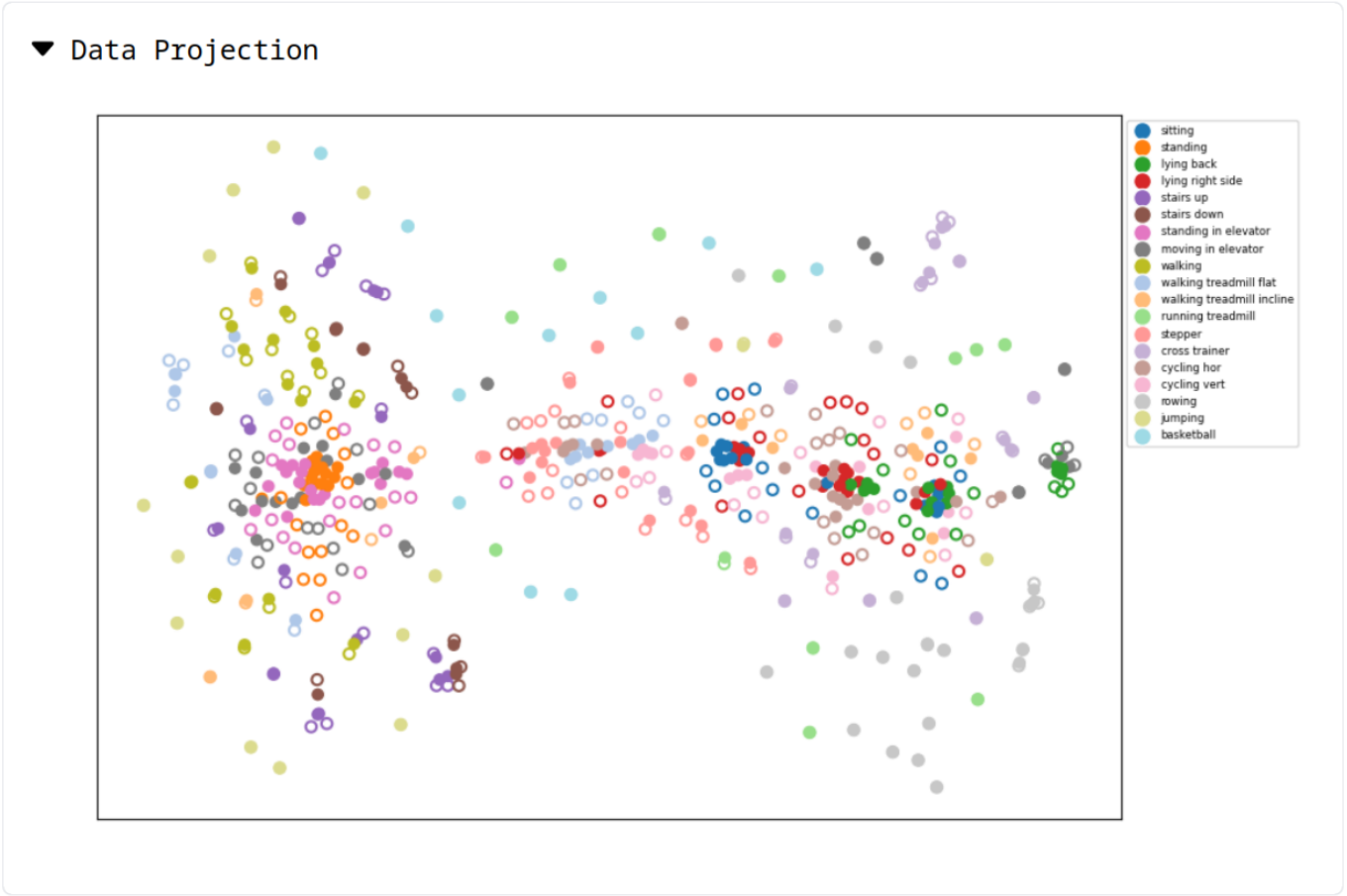


Figure 3.7: Projection of original and generated signals from the DSADS dataset in 2D space using t-SNE.
Filled circles are original signals, while unfilled circles are generated ones. Each data point is colored according to its class.

convolutional layer to visualize and selecting a specific input channel. As illustrated in figure 3.8, each invididual kernel is depicted after the options menu.

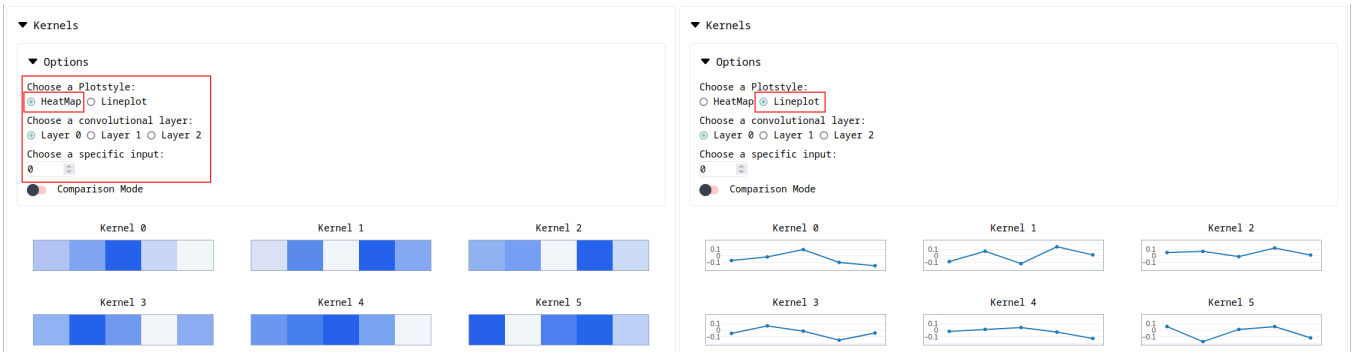


Figure 3.8: Exploration of kernel visualizations in both heatmap and line plot mode in TimeLens-Web with additional options. Kernel visualizations are shown below the options.

To observe the application of kernels on real training data, another option is the examination of features maps, as illustrated in figure 3.9. The options menu still provides the option to select a specific convolutional layer, along with a button to generate a new sample. This is crucial since feature maps depend on the input of a CNN. That input sample can therefore be viewed at the top positioned above all feature maps for each kernel, which are represented as line plots.

Figure 3.9: Feature map exploration given the depicted input sample. The sample that is used to generate the feature maps is shown above the extracted feature maps.

The last tab called Activation Maximization offers three possibilities: performing activation maxi-

mization (AM) for a single kernel, for a whole layer or producing an Activation Atlas.

As mentioned earlier, TimeLens enables the computation of AM with different inputs, which first have to be selected in the web application. The process involves choosing an input signal and specifying a particular layer and kernel for maximization, as illustrated in figure 3.10. Since the options and selection of input remain consistent (except choosing a specific kernel) between AM layer generation and the Activation Atlas, they won't be presented again.

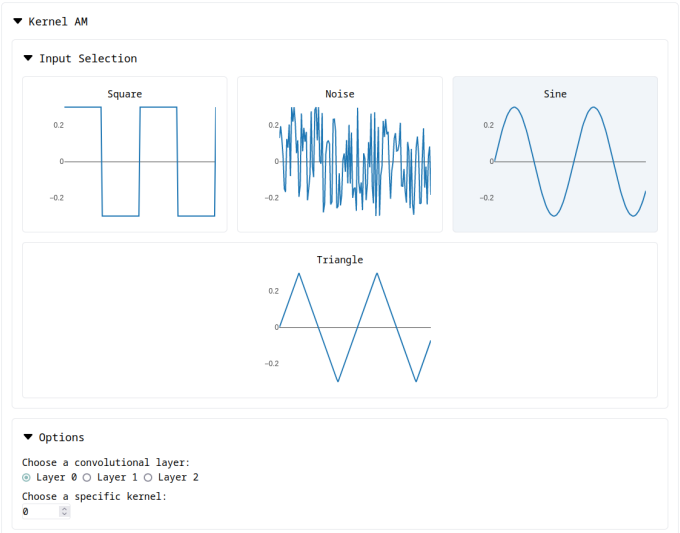


Figure 3.10: Selection of AM input and configuration options before starting the process

After configuring those options, the activation maximization (AM) process initiates, producing signals that maximize activation for a specific unit by leveraging gradient ascent from equation 2.34. An exemplary result of this process can be seen in figure 3.11.

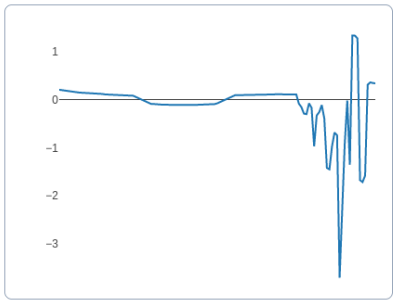


Figure 3.11: Exemplary result of activation maximization performed within TimeLens-Web

Maximized input signals for an entire layer follow a similar generation process to those for a specific unit. They are visualized similarly to feature maps for a complete layer and are therefore not

explicitly depicted here.

A primary focus of this thesis is to investigate how particular data augmentations impact the internal mechanisms of a CNN. To support this analysis, the web-based tool includes a comparison mode for two different models. By clicking the plus button at the top, a new model window opens alongside, providing access to all the features previously described. As illustrated by figure 3.12, this functionality allows, for instance, the comparison of a model with jittered data and a model with permutated data, showcasing their dataset projections side by side. This side by side comparison greatly facilitates the understanding of changes induced by various data augmentations.

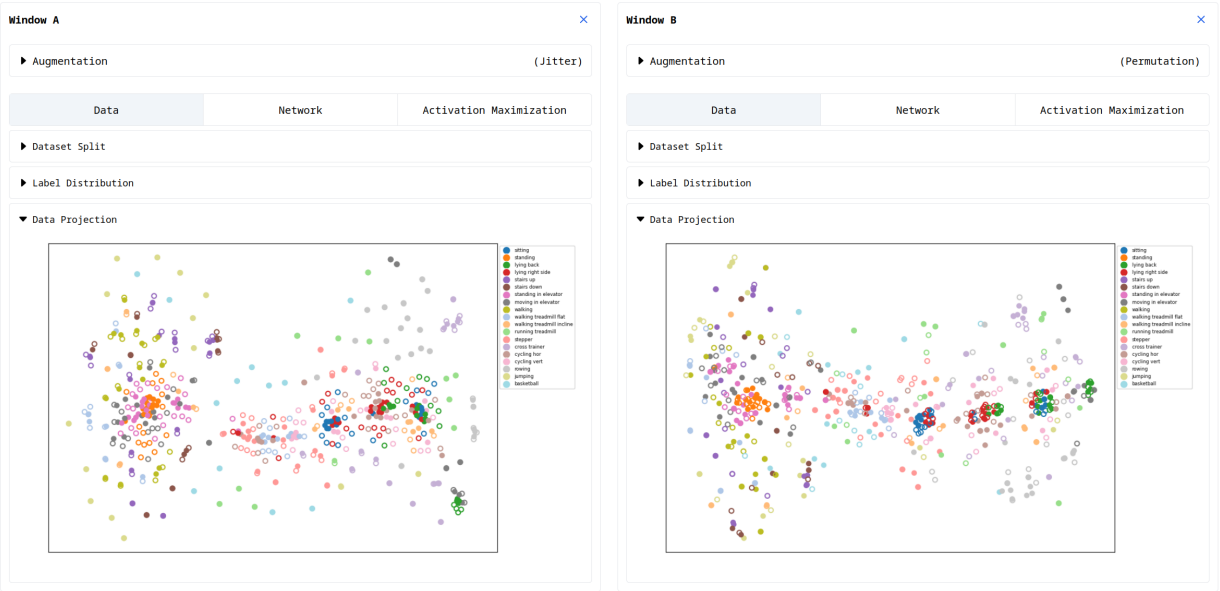


Figure 3.12: Side by side comparison of dataset projections for two different data augmentations. The dataset projection of jitter and permutation can be seen on the left and right, respectively. The included legend on the side of each plot assigns a color to each class of the dataset.

To further enhance the comprehension of the effects of data augmentations, a comparison mode is also available for various visualizations. For instance, when opening feature maps for two models, enabling the comparison mode assesses each feature map of the first model against the matching feature map of the second model, calculating the difference between them using the euclidean distance metric. After computing all differences, the feature maps are sorted by their descending euclidean distance, allowing users to quickly identify which feature maps changed the most. Both the option for the comparison mode and the kernels sorted by their descending Euclidean distance are illustrated in figure 3.13.
However, it is crucial to note that a fundamental assumption for this calculation is that the kernels of both models are in the same position. The explanation of why this assumption holds will be detailed further in section 3.4.

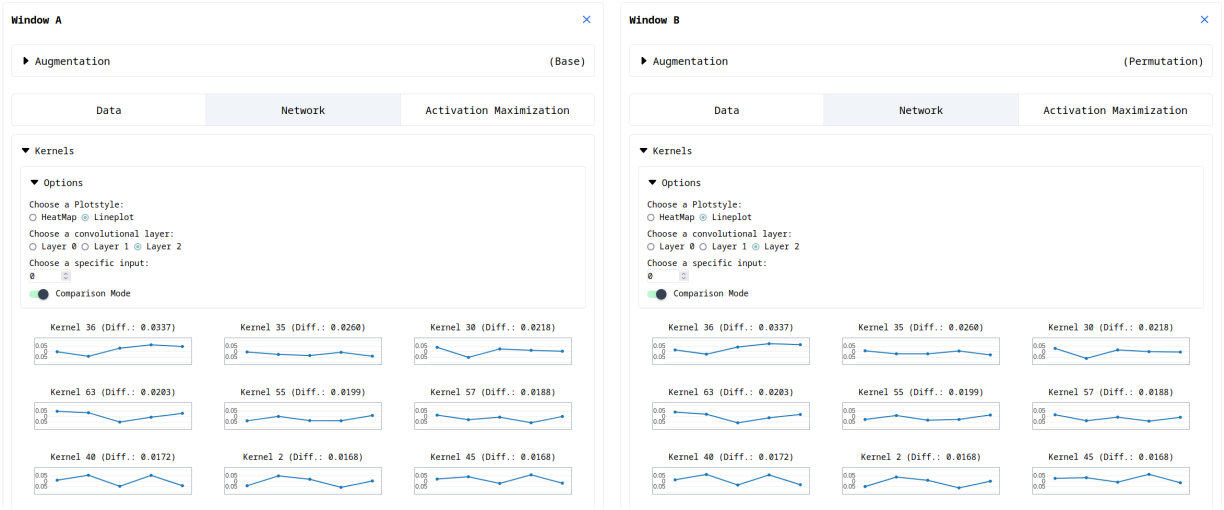


Figure 3.13: Side by side comparison of kernels for the base model versus the model with permuted data. The enabled comparison mode leads to a calculation of the euclidean distance between two matching kernels shown above the plot.

3.3 Activation Atlas

An Activation Atlas tries to combine the visualization of a whole layer into a single plot by performing AM for each unit and combining them into a meaningful representation. An illustrative example can be seen in figure 3.14, with the generation and therefore the visualization itself explained in the following.

The visualization of an Activation Atlas similar to Yoshimura, Maekawa, and Hara [47] first includes the computation of AM for all units of a specific convolutional layer. Next, the resulting maximized activations are projected into 2D space using t-SNE [30]. t-SNE is chosen over UMAP [32] or principal component analysis (PCA) due to its ability to produce representations that are both well separated and consistent across various data augmentations. When compared to PCA, which often resulted in heavily clustered representations, and UMAP, which led to inconsistent representations when applied to different datasets or even the same dataset again, t-SNE stands out. Its proper separation and consistency make t-SNE a preferable choice, as it allows for more meaningful representations that can be easily compared to each other.

Since the Activation Atlas is a visualization technique including the dimensionality reduction of generated samples in the center with additional exemplary plots around the edges, it is essential to first select a few representative samples. To ensure a diverse representation of samples, K-Means [31] is employed. This involves computing 14 centroids to match the required number of samples around the edges. Figure 3.15 illustrates 14 computed centroids depicted as orange dots for an exemplary convolutional layer and its projected maximized activation signals.
Afterwards, for each centroid the sample with the smallest distance was extracted. To enable a

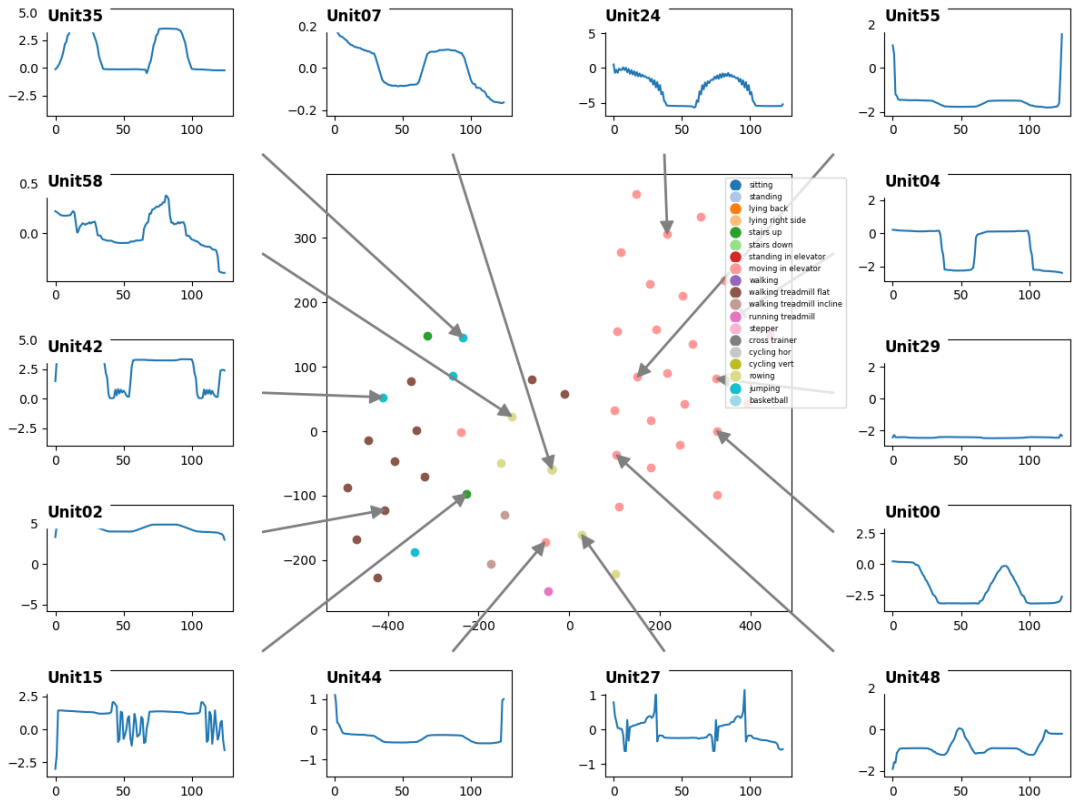


Figure 3.14: Generated exemplary Activation Atlas. The middle area depicts the dimensionality reduction of generated maximized signals. Around the edges several exemplary signals are shown.

completely dynamic generation of the visualization with proper placement of samples around the edges, the extracted signals are assigned a spot around the center by leveraging the hungarian algorithm to optimize the sum of distances between the plot coordinates and the coordinate of the centroid.

To enable additional interpretation of the dimensionality-reduced samples and their positioning inside the 2D space, each maximized signal is compared to the training data to find the signal with the lowest euclidean distance. The label of the found signal is utilized to approximate a label for the generated signal and used in the visualization process. When coupled with the previously described AM process for the entire layer, this approach creates a visualization that offers a comprehensive understanding of the features extracted by an entire convolutional layer.

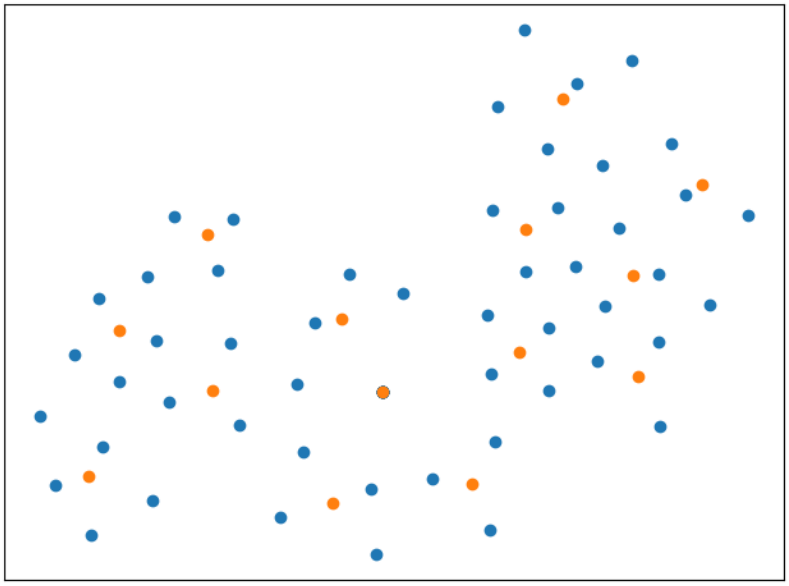


Figure 3.15: Calculated centroids (orange) for the same projection as in figure 3.14

3.4 Reproducibility

As a critical aspect of this thesis involves comparing the effects of different data augmentations on the performance and internal representations of models, it is crucial to ensure that the models themselves are comparable. To achieve this, the usually non-deterministic training process of neural networks needs to be as deterministic as possible. Another primary reason for proper reproducibility is the comparison of feature maps and kernels mentioned in section 3.2. During normal training, random weight initialization and the random order of training samples lead to different models that may perform similarly but do not function internally in the same way. Even when training the same model without any data augmentations, finding the matching kernel (the kernel that extracts the same feature) for each kernel inside a convolutional layer proved challenging. Therefore, to facilitate the comparison and comprehension of changes inside of a CNN, several reproducibility measures have been implemented. As ensuring reproducible training with PyTorch is not a trivial task, the measures will be described in detail in the following.
To ensure proper control over sources of randomness, the Python packages numpy, random and torch were all seeded using the same seed. Another potential source of non-deterministic behavior are CUDA convolution operations, which benchmark different convolution algorithms to identify the fastest one. To eliminate this variability, the benchmarking feature was disabled. However, even with a consistent convolution algorithm, the algorithm itself might introduce non-determinism, leading to variation in training runs. To address this, deterministic algorithm are enforced for both CUDA convolution operations and PyTorch algorithms.
For CUDA versions above 10.2, when training on the GPU, debug environment variables must be set to ensure that bit-wise operations produce reproducible results across different CUDA toolkit versions.

Additionally, each data loader used in the training process is initialized with a seeded worker to guarantee reproducibility, as these seeds are otherwise set randomly and are not affected by previous seeding processes.

3.5 Network Similarity

Assuming that kernels in different neural networks match their location under the precautions explained in detail in section 3.4, it becomes feasible to compute a metric for quantifying the similarity of two networks. The norm will be referred to as network similarity in the following.
To compute network similarity, access to all convolutional kernels from both networks is required. TimeLens, a package introduced in section 3.1, can be employed for this task. As the absolute value of the metric might not convey meaningful insights, it is recommended to compare the networks being evaluated to the same base model. This base model could be a model trained without any data augmentations, serving as a reference point point for assessing the changes made to the network through training with data augmentations.

The metric can be computed with the following formula under the previous assumptions, which include the same amounts of channels and kernels for both networks:

$$NetworkSimilarity(A, B) = \sum_{l=1}^{L_A} \sum_{c=1}^{C_{A,l}} \sum_{d=1}^{D_{A,l}} ||k_{A,l,c,d} - k_{B,l,c,d}|| \tag{3.1}$$

Here, L is the number of convolutional layers for a specific model, C is the number of channels for a specific convolutional layer and D is the number of kernels in a specific convolutional layer. $k_{A,l,c,d}$ represents the weights of a kernel in model A, at layer l, in channel c and at kernel index d. Using both kernels, the euclidean distance is calculated as a similarity metric and then summed up over all kernels, channels and layers. To interpret the metric, it is crucial to consider that comparing it with other computations of the same metric will yield the most insights. Additionally, since Network Similarity is a sum of distances, the lower the result, the more similar the compared model is to the base model.

As the single value retrieved by the metric (the sum) might not be as meaningful, it is recommended to save all values along the way. These values can be used to calculate summarizing metrics like the mean, the standard deviation and minimum and maximum values.

4 Experiments and Results

This chapter will cover the experimental setup, detailing the parameters and penalties for activation maximization, the preprocessing of datasets and the architecture for CNNs utilized. Furthermore, it will present all quantitative and qualitative results, including dataset projections, activation maximizations and network similarity.

4.1 Experimental Setup

Model Architecture To ensure good performance for each dataset, two different models were created, as both datasets demand distinct approaches in terms of model architecture.
For the FordA dataset, the model architecture comprises three convolutional layers, each with 16 kernels. Each kernel is of size five, applied with a padding of two to retain information at the edges (see equation 2.12), and a stride of one. Additionally, max pooling with a kernel size of two is applied as the pooling function. The fully-connected segment consists of one layer with 25 nodes. Following the linear layer, the NN has two output nodes. The model architecture was selected by training several models with different architectures based on their performance on the test dataset. To maintain simplicity, all NNs are trained with cross-entropy loss (equation 2.13), resulting in the model having two output nodes instead of one, in combination with thresholding for classification. For faster convergence, batch normalization is included.

Given that the DSADS dataset is multivariate, the first convolutional layer requires an input size of six channels. Adopted from the work of Yoshimura, Maekawa, and Hara [47], the architecture involves three convolutional layers, each with 64 kernels of size five. A stride of one and a padding of two are applied to prevent information loss when convolving near the signal edges. The fully-connected layer is more complicated, consisting of two linear layers, each with 256 nodes. Max pooling with a kernel size of two is incorporated, and, consistent with the earlier network, batch normalization is utilized to enhance convergence.

Training Parameters All training parameters can be found in table 4.1. Both networks are trained with cross entropy loss, as defined in equation 2.13. The prediction performances for models were computed by computing the mean over ten seeds. When training was finished, the weights of the epoch with the highest training accuracy were loaded.

Optimizer	Adam
Learning Rate	10^{-4}
Batch Size	32
Epochs	200
Patience	10
Hardware	NVIDIA GeForce RTX 3090TI

Table 4.1: Chosen values for Training Parameters for CNNs employed in this thesis.

Activation Maximization Activation Maximization (AM) is the process of generating signals that maximize the activation of a certain NN unit by leveraging the gradient (described in detail in section 2.4). However, as indicated by Yoshimura, Maekawa, and Hara [47], employing traditional AM methods on time series data results in ambiguous and noisy signals when directly leveraging the raw values of a time series sample. This observation is particularly applicable to acceleration-based data in their study. Their research revealed that the utilization of various regularization techniques, including the Lp norm, total variation, extreme activation penalty (EAP), and clipping, yielded the most favorable quantitative outcomes when assessing the similarity of generated signals to the original training data. Formulas for all those penalties can be found in section 2.4. Moreover, qualitatively, these techniques appeared to generate smooth and interpretable signals. Therefore, all penalties will be applied for the experiments using equation 2.33. As the penalties are applied after several other steps, the entire process will be described sequentially. In brief, the algorithm consists of three steps: initialization, activation retrieval and updating the input.
To iteratively update a signal, we need a starting point to improve on when applying gradient ascent. Various input signals can be used, which are summarized in figure 4.1. A common input often used

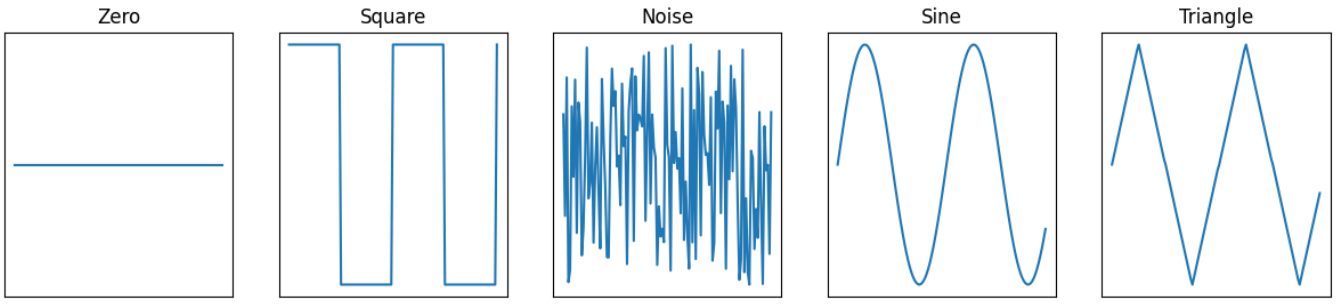


Figure 4.1: Line plots of various different inputs for Activation Maximization

for images is random noise [48]. The time series inputs chosen here are different representative examples and consist of zero, square, noise, sine and triangle input [47]. Cyclic inputs were generated to include approximately two cycles, with a maximum amplitude set at 0.3 for all inputs. The effects of different AM inputs on the final generated signal and whether they matter will be covered in a later chapter.

Afterwards the AM process is initialized with a learning rate $\eta = 0.001$, which is decayed by multiplying it with $\gamma = 0.999$ after every of the 5000 update steps performed. After initialization, the

operation is computed for a specific node by performing an AM step for each iteration. The step itself retrieves the activation for a specific unit by feeding the input signal through the network and then clipping each activation value. The clipping value was determined based on the original dataset and calculated by taking the mean plus two times the standard deviation. Following the clipping procedure, the activations were condensed to a singular value by computing the mean of all activations. Subsequently, penalties were applied in the sequence of EAP, total variation, and Lp norm. The threshold employed for EAP was identical to the value used for clipping. The weight used for all penalties was 0.1. Finally the gradient was computed and used to update the input time series as seen in equation 2.34.

As described in section 3.3, the selection of examples to depict within the Activation Atlas involves leveraging K-Means and euclidean distances. While this method performs well when observing a single Activation Atlas, some challenges arise in the intercomparability between Activation Atlases. Small deviations in the network, such as those caused by data augmentations, might shift the distribution of data, leading to a slightly different 2D representation computed by t-SNE. This results in different K-Means clusters and, consequently, different visualized kernels around the edges of the illustration. To facilitate easier and more meaningful comparisons of Activation Atlases, the representative kernels of a base visualization are saved and used to choose the other samples. This ensures that the same representative kernels are compared. The perplexity for t-SNE was set to 30.

In section 4.2.1, six different models each trained with a specific data augmentation applied to its training dataset are compared based on their Activation Atlas. The process of generating these is described in the following. The initial model is trained on a dataset without any data augmentations, whereas the remaining five are each trained with a distinct data augmentation applied. Each model trained with an augmented dataset will be compared to the base dataset by describing alterations in their generated maximized activation signals, utilizing the Activation Atlas. As already explained in the previous chapter, the exemplary visualizations generated around the center are created based on the same kernel positions. Even when applying different data augmentations, this ensures a stable foundation for the comparison of Activation Atlases. However, it is crucial to note that although the same kernels are depicted, they are not positioned identically. Placing them in the same position would result in overlapping arrows due to slight variations in the signals leading to a distinct t-SNE projection.

Data Augmentations The selected data augmentation methods include jitter, scaling, slicing, permutation and magnitude warping (described in detail in section 2.3.1). Similarly to Iwana and Uchida [23], these data augmentations methods have been selected because they are general methods that can be applied to any type of time series. Furthermore, to streamline the process no data augmentations methods were used that required external training like generative models.

The standard deviation necessary for sampling the Gaussian noise for jittering, as shown in equation

2.26, was set to 0.03. As it is Gaussian noise, the mean is set to zero. A small value was chosen here for the standard deviation to ensure that the added noise is not too extreme and preserves the original structure of the data.
In the case of scaling, the same scaling factor $\alpha \sim \mathcal{N}(1, \sigma^2)$ is employed for the time series. By multiplicatively applying the scaling factor, as described in equation 2.27, with a standard deviation of 0.1 and mean of 1, this ensures the generation of signals that do not deviate excessively from the original samples and are still within their domain ranges.
When applying the slicing data augmentation, the size of the extracted signals in equation 2.29 were set to 90% of the original length of the data samples. 90% was chosen here to extract samples that are slightly shorter than the original samples to ensure that each sliced segment still captures most of the temporal context of the original data. This helps in introducing variability without losing the overall structure of the samples.
In the experiments with permutation, equal-sized segments were used, meaning that each sample was divided into N segments of size $\frac{T}{N}$ and then randomly shuffled. As dividing each sample into segments and shuffling them introduces randomness without completely disrupting the temporal relationships with the segments, a segment size of five was chosen to maintain a balance between introducing variability and maintaining local patterns.
The two hyperparameters for magnitude warping in 2.28, σ and I, were set to 0.2 and 4, respectively. A standard deviation of 0.2 ensures a moderate level of warping, preventing excessive distortion. The parameter I controls the number of interpolation points, and a value of 4 maintains a balance between smooth warping and introducing variations.
In summary, the chosen parameters aim to introduce variability in the data while ensuring that the augmented samples remain realistic and representative of the underlying patterns in the time series data.

It is also important to evaluate whether results and comparisons derived from the evaluation of individual augmentations remain consistent when two data augmentations are combined. Therefore, jittering has been combined with slicing and the permutation process. The order of application, in this case, does not affect the outcomes, as these augmentations target different domains of transformation. Nevertheless, jittering has been applied first in this combined approach. The parameters chosen for the combined augmentations are equivalent to those used in their individual application.

Datasets The FordA dataset was pre-processed with the following steps: Missing values were filled with the last valid value before the missing slot. Since each row corresponds to an individual sample, the labels are subsequently removed from each row and saved separately. Next, each value is z-normalized using the following formula:

$$z_i = \frac{x_i - \mu}{\sigma}$$

where z_i is the standardized value at position i, x_i is the original value at time step i, μ is the mean and σ is the standard deviation. Given that the dataset is already divided into training and test sets, these sets will be utilized directly with their given splits.

Regarding the DSADS dataset, as an initial preprocessing step, only six time series are extracted from each sample, specifically all acceleration signals from the right arm and the left leg. This simplification aligns with the approach outlined by Yoshimura, Maekawa, and Hara [47]. Since each sample in the dataset is initially aggregated into a single file, this file includes the label, patient information, and the actual measurements. When loading the dataset from the file, the label and patients are extracted from the data and saved separately. Subsequently, to enhance robustness by directing the network's attention to the relative intensity of accelerations, the acceleration data undergoes a transformation from $\frac{m}{s^2}$ to g using the following formula:

$$z_i = \frac{\frac{x_i}{9.806665}}{3}$$

Following this transformation to g, the data is further normalized by dividing it through its maximum value in both directions (3), ensuring that the g values fall within the range of $[-1, 1]$. Given that the dataset divides five minute signals into smaller time series, utilizing a random shuffle of the data to create a training and test dataset could result in the leakage of related test data into the training dataset. Such leakage would compromise the validity of any evaluations performed on the model. However, the authors have already specified the use of two patients for testing, with the remaining six designated for training. This recommendation is adhered to in this thesis [3].

As the effects of data augmentations on prediction performance appears to diminish with an increase in training data, the training datasets of both datasets were not fully utilized. Initially, the training data was shuffled, and then the first 300 samples were selected for the training process of both models for the DSADS and FordA datasets. Additional details can be found in section 4.3.2.

4.2 Qualitative Results

4.2.1 Activation Maximization Results

Application of AM to different convolutional layers

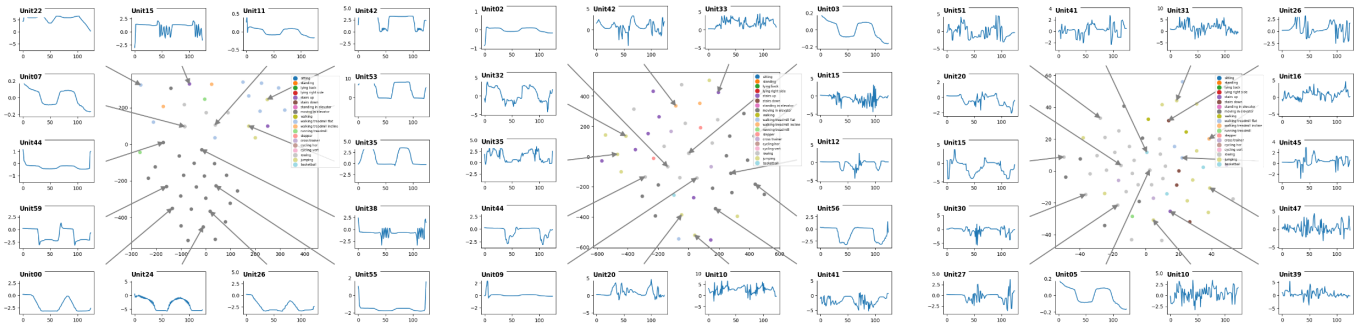


Figure 4.2: Activation Atlas generated for different convolutional layers on a subset of the DSADS dataset. The center of each plot illustrates the dimensionality reduced maximized activations. Around the edges are exemplary signals.

Figure 4.2 illustrates the atlas containing examples of generated signals from convolutional layers ranging from the first to the third layer. These visualizations were created with a sine wave, as seen in section 4.1, as an input signal and a CNN trained with no data augmentations on a subset of the DSADS dataset. Maximizing signals from the first convolutional layer appears to extract basic features such as slopes or simple waveforms. Examples near the bottom (unit 0, 24, 26, 55) of the first Activation Atlas seem to focus on negative values, while those in the upper right corner (unit 42, 53, 35) seem to extract positively valued waveforms.

On the other hand, samples generated by applying AM to the third convolutional layer appear more complex and contain high frequencies. While kernels like unit 15 are maximally activated by more complex waveforms than in layer 1, other signals for kernels like unit 10 or 47 depict waveforms with high-frequency components.

The visualization in the middle and therefore the Activation Atlas of the second convolutional layer, seems to be a middle ground between the first and third layer. It contains both simpler features such as those extracted by kernel 2, 9 and 44 and kernels that extract more complex features like number 10 and 15.

In conclusion, these findings suggest that the first convolutional layers tend to extract basic features like slopes and simple waveforms. Subsequent layers, on the other hand appear to extract signals

with high-frequency components and more complicated forms in general. This aligns with other research mentioned in section 2.4.
Given that the second convolutional layer appears to offer a balanced mixture of basic and complex features, it is used for all future AM calculations.

Computation of AM with different input signals

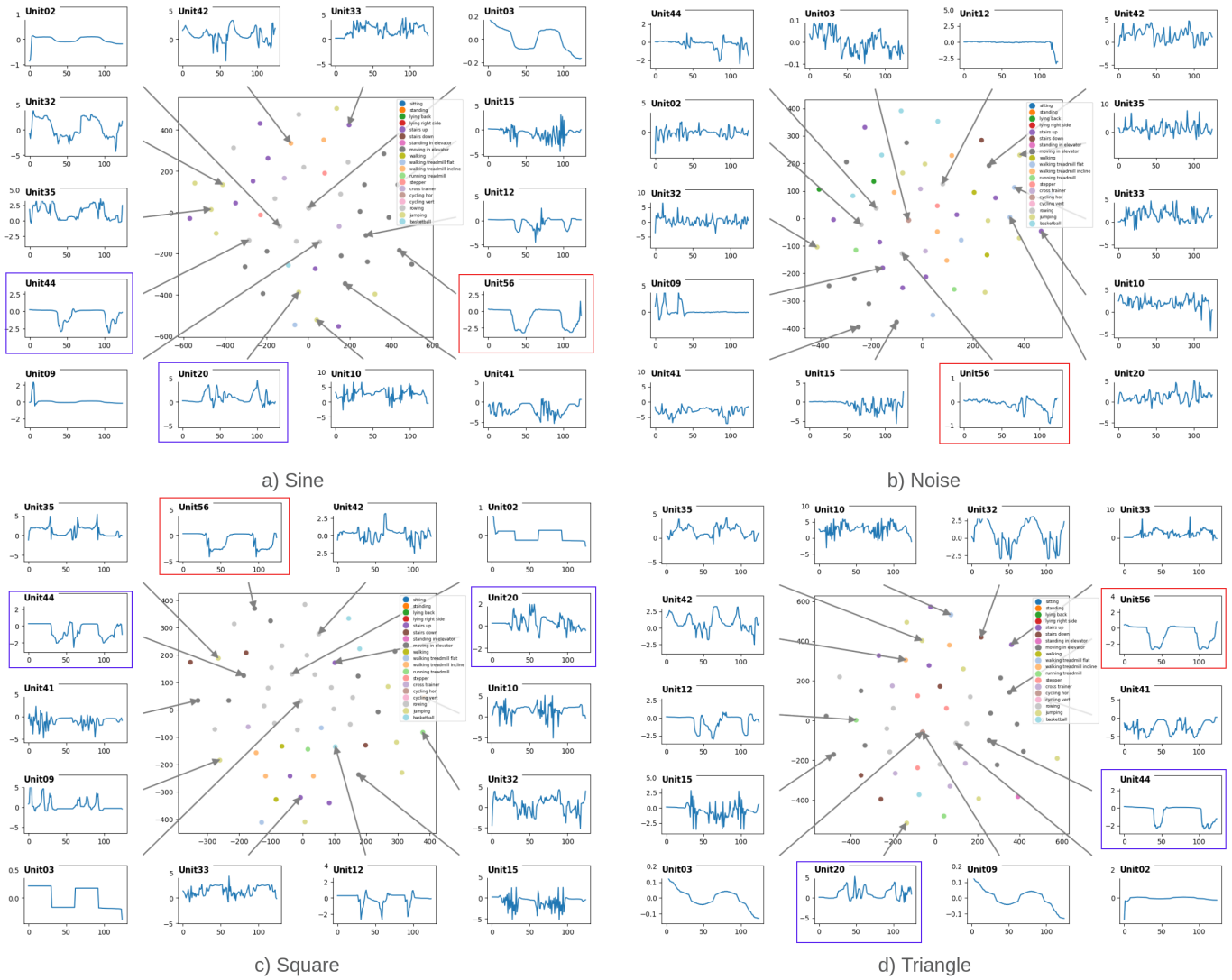


Figure 4.3: Activation Atlas generated for different input signals for the second convolutional layer of a CNN. The network was trained without any data augmentations on a subset of the DSADS dataset. The different input signals can be seen in section 4.1

An essential parameter for AM is the initial signal that is optimized according to equation 2.34. Several options have been presented in section 4.1, and the generation of an Activation Atlas with each input is illustrated in 4.3. These visualizations were generated for the second convolutional layer of a CNN trained without any data augmentations on a subset of the DSADS dataset.

Figure 4.3 presents four Activation Atlases, each generated with a different input signal. While sine, square and triangle express similar shapes, as seen in section 4.1, the generated random noise signal does not exhibit any cyclic behavior and is therefore different. It is evident that signals generated utilizing the noise input differ from those generated by sine, square or triangle. When observing kernel number 56, marked with a red rectangle, the generated signals from sine, square and triangle resemble a simple waveform with mostly negative values. The AM with a noisy input, on the other hand, generates a signal that completely misses the first decrease apparent in the other signals. That is just one of many examples, where the generated noise signals differ significantly from the other ones.

Compared to the sine and triangle input, the square input contains regions with a vertical pitch. This vertical pitch and its effects on the gradient ascent process when updating weights, might be the reason for a higher amount of induced noise in the generated signals for the square input signal. Kernel number 20 and 44 are marked with a purple rectangle in a, c and d in figure 4.3. In comparison to sine and triangle, the square signals contain short high-frequency components. This effect can also be seen in kernel 56 marked with a red rectangle.
Additionally, the input signal based on Gaussian noise seems to consist mostly of high-frequency components. This is probably caused by the high amount of noise in the input signal impeding a smooth maximization process.

In conclusion, similar input signals appear to produce similar maximized signals. However, entirely different input signals, such as noise and sine, seem to result in significantly different signals. There, the activation maximization (AM) process applied in this thesis is not invariant to the input signal. While it is robust to small deviations in the input, the choice of input signal can have a substantial impact on the generated maximized signals.
Given that the sine input produces less noisy activations compared to noise and is similar to square and triangle, it is chosen as the input signal for the remainder of this section.

Impact of data augmentations on extracted features

Following the selection of an initialization signal and convolutional layer, the influence of various data augmentations can be observed in figure 4.4. Beginning with jitter, it can be observed hat the magnitudes of certain signals are influenced by the data augmentation. Two instances, namely kernel 10 and 12, are highlighted with purple frames. In their cases, the activations are reduced

to approximately half of the original value. Another noteworthy effect is the introduction of high-frequency components for some kernels. Three regions are delineated with red rectangles. While unit 15 and 41 already contained high-frequency components, the data augmentation introduced some new ones in kernel 56.

Moving on to permutation, a data augmentation that shuffles random segments of a time series, it appears to cause a different effect on the AM process. Instead of introducing or amplifying high-frequency components, it seems to smoothen out some high-frequency areas. This effect is observable in many kernels, accentuated here by a purple outline. Additionally, it seems like permutation caused a shift of the peak in kernel 56. While the area in the first valley is relatively flat in the base model, the maximized signal for the augmented model exhibits a peak. This peak might be shifted from the end of the signal due to the permutation applied to the training data.

In line with permutation, the Activation Atlas for scaling depicts signals with smoothed high-frequency components, indicated by a purple rectangle in kernel 10. However, while smoothing certain high-frequency components, it also appears to introduce new ones in kernel 42 and 32. Red rectangles draw attentions to areas in kernel 41 and 12, where augmenting the training samples resulted in maximized signals that have a higher or lower magnitude compared to the base model. While kernel 12 got scaled down to approximately half of its original value, kernel 41 exhibits peaks that appear to double in their amplitude.

High-frequency components in kernel 15 and 10 appear to be amplified when examining the effects of the slicing data augmentation in plot e). These high-frequency components are present in the original model, but have both increased in their length and amplitude. Sections near the edges of generated signals, emphasized by a purple rectangle, exhibit alterations for some signals, often becoming more flat or introducing new patterns. The extraction of a subset of a signal by augmenting with a slicing data augmentation frequently results in data loss near the edges, potentially contributing to these changes.

The final distinct data augmentation discussed here is magnitude warp. As depicted in equation 2.28, it performs transformations in the magnitude domain by warping the magnitude of a time series using a smoothed curve. Notably, not caused by any other data augmentation thus far, are the comprehensive modifications of maximized signals for a few kernels. Two examples are the highlighted purple kernels 35 and 10. Especially in the case of kernel 10, the signal is not easily recognizable when compared to the base model. Signals highlighted in red exhibit both increased and decreased magnitude, sometimes within the same signal. An instance is for example the signal for kernel 12, where the depth of the valley increased while the subsequent values got dampened to some extent.

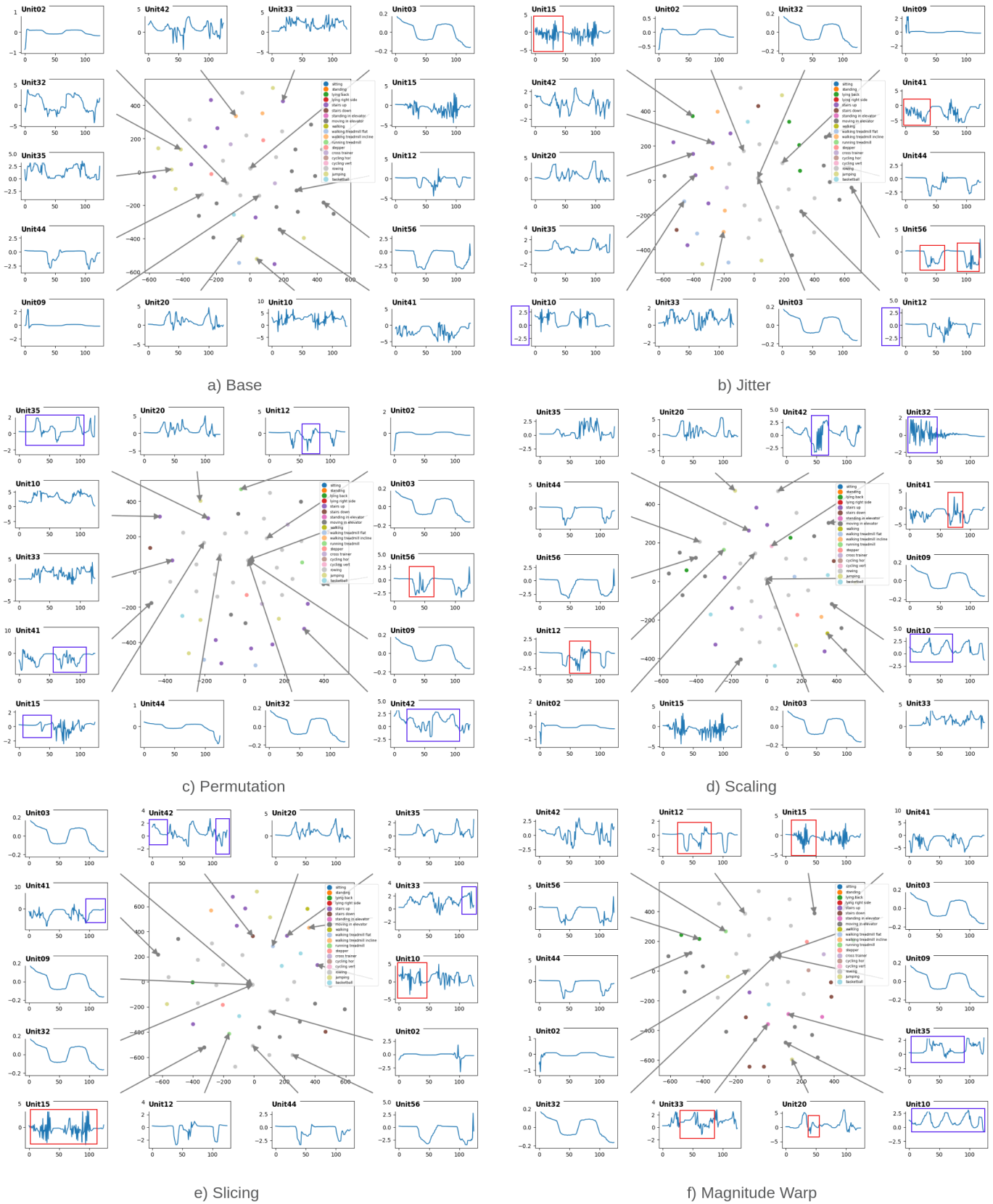
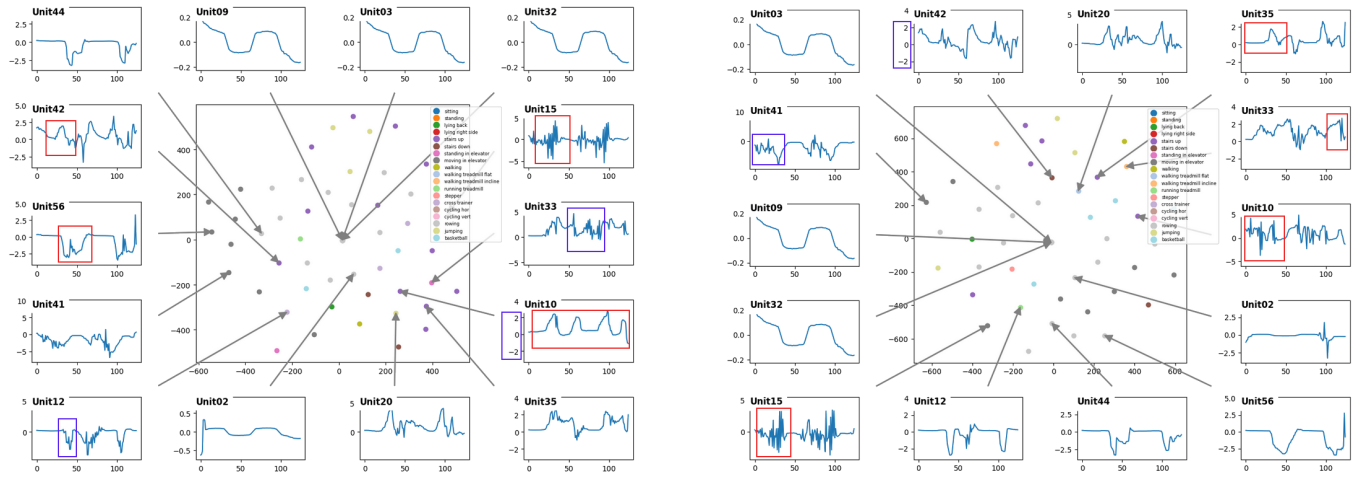


Figure 4.4: Activation Atlas generated for different data augmentations for the second convolutional layer of a CNN. The network was trained on a subset of the DSADS dataset.

Having observed the changes induced in maximized signals when applying a single data augmentation to a dataset, the next step involves examining the effects when two data augmentations are applied. As there are too many possible combinations to cover in this thesis, jitter combined with permutation and jitter combined with slicing have been chosen as representative examples. Figure 4.5 illustrates both Activation Atlases.

The initial combination of data augmentations discussed involves jitter and slicing. The effects of jitter, as described earlier, include the introduction of high-frequency components and the reduction of signal magnitude. Both effects are observable in the signals for kernel 41 and 42. However, signals from kernels that were previously compressed no longer exhibit that compression (unit 10).
The earlier observed amplification of existing high-frequency components caused by slicing can be observed together with modifications of signals near their edges.
Both augmentations appear to retain their effect. However, the scope and intensity and modifications were also influenced by the combination.

Regarding the combination of jitter and permutation, all the effects described previously can be identified in the combination of both data augmentations. Purple rectangles emphasize the earlier effects of jitter, including the reduced magnitude on some kernels and the introduction high-frequency components. The introduction of high-frequency components can be observed in kernel 12 and kernel 33. The reduction of magnitude, earlier apparent in kernel 10 and 12, seems to be present only for kernel 10 in this case.
Effects observed in the earlier Activation Atlas for the application of permutation are highlighted with red rectangles. Compared to only applying permutation, the application of both jitter and permutations seems to cause more shifted peeks in kernel 56 and kernel 15, while smoothing appears to exist to a similar degree.
Upon observing these changes, it becomes apparent that both data augmentations seem to retain their effects on the maximized signals to some degree. However, the introduction of high-frequency components seems to be mitigated to some extent by the smoothing effect of permutation.



a) Jitter + Permutation

b) Jitter + Slicing

Figure 4.5: Activation Atlas generated for two different combinations of data augmentations for the second convolutional layer of a CNN. The network was trained on a subset of the DSADS dataset.

4.2.2 Exploration of Augmentation Impact on Kernels

The exploration of feature maps and kernels when comparing two neural networks already offers a vast amount of possibilities. Comparing multiple neural networks to each other using this approach would be an extensive task, beyond the possible scope for this thesis.
Nevertheless, to demonstrate the utilization of the comparison feature of TimeLens-Web one specific example will be examined in the following.

To achieve this, alterations in kernels resulting from applying data augmentations to the training data are observed in comparison to the base model. This procedure is performed for jitter, permutation and their combination. Here, we want to address whether the application of a combination of jitter and permutation yields a similar effect on the kernels as their individual applications.

Figure 4.6 visualizes kernels for the base model and three other models with applied data augmentations. These kernels are extracted from the final convolutional layer, showcasing only the twelve kernels with the greatest distance compared to the base model. The Euclidean distance relative to the base model is indicated above each kernel. Further details about the visualization process can be found in section 3.2.

Figure 4.6: Visualization of kernels for the third convolutional layer of four different models trained with various data augmentations. All kernels are of length 5 and are depicted as a line plot. The euclidean distance in comparison with the base model is displayed above all models that have been trained with a data augmentation. Kernels are sorted by their euclidean distance compared to the base model.

Kernel 62 and 46 both appear in the kernel list of jitter and the combination of jitter and permutation. They have been highlighted with a red rectangle for enhanced visualization. However, these are not the only kernels evident in the kernel visualization for both data augmentations. Kernel 36, 45 and 57 appear in all three visualizations, indicating that these kernels were significantly affected by both jitter and permutation. These specific kernels might be more prone to be affected by data augmentations.
When examining the kernels most affected by permutation, it can be noted that none of those exclusively changed by permutation appear in the list for jitter and permutation. A possible explanation might be that jitter has a more pronounced input on this specific input channel in the third convolutional layer compared to permutation as seen in kernel 62 and 46.

4.2.3 Augmentation Impact on Datasets

To further understand the impact of data augmentations on the actual data, a projection into 2D space, as earlier detailed in section, 3.2 can be insightful. Figure 4.7 illustrates projections for various data augmentations, starting with jitter in the upper left corner. In this figure, a subset of the DSADS dataset is visualized alongside its augmented data. Hollow points represent synthesized

samples, while solid points correspond to the original data points. The colors indicate different classes, though the legend is omitted for space considerations. If specific class assignments are necessary, a legend of classes can be found in figure 3.7.

When visualizing generated patterns through dimensionality reduction, differences between the patterns generated by various data augmentation methods become apparent. In all projected datasets, generated samples exhibit noticeable deviation from their original counterparts near a horizontal line through the middle of the dataset. This level of deviation can be estimated by observing the distance between filled and non-filled samples of the same color in a specific vicinity.

In each plot, certain discriminative regions have been identified and marked with a red rectangle. In the current visualization of jitter, it can be observed that generated samples do not deviate much from their original signals. Although there are larger spreads around the horizontal center line, the dimensionality reduced original samples near the edges are essentially the same as the generated ones. This is evident by the fact that the filled and unfilled dot often perfectly overlap.

On the other hand, the use of permutation data augmentation appears to generate samples that diverge more from their original samples. Multiple instances around the edges show augmented samples distanced from their original counterparts, indicating a considerable spread. In this case, permutation seems to cover a larger portion of the input space.

Scaling yields a result similar to jittering. However, it is noticeable that samples near the horizontal center line are more closely clustered. This suggests a higher similarity between generated and original samples in this region. Nevertheless, samples around the edges are also tightly grouped and overlap with counterparts.

When projecting samples and their sliced counterparts into 2D space, the resulting visualization exhibits overlapping generated signals. There are several instances, where a generated signal from a particular class closely aligns with a generated signal from another class, resulting in an overlap in the visualization. Additionally, the overall spread appears to be higher than that observed in jitter or scaling visualizations.

Lastly, the last singular data augmentation magnitude warp does not appear to alter the data significantly. The visualization closely resembles other magnitude based random transformations such as jitter and scaling, showing minimal deviation in samples near the edges and an average spread near the horizontal center line.

The combination of jitter and permutation in a single dataset produces a projected visualization showing similar results to those generated by permutation alone. Deviations by generated samples from their original counterparts are notable, particularly near the edges. Additionally, there appears

to be a larger variance in general. As observed earlier in the jitter plot, jittering does not seem to introduce distinctive features when compared to other data augmentations and, therefore, the difference is mainly driven by permutation.

Figure 4.7: Datasets consisting of a subset of the original DSADS dataset and differently augmented data projected into 2D space using t-SNE

4.3 Quantitative Results

In the subsequent two subsections quantitative results will be presented. Firstly, various neural networks will be compared by using the metric introduced earlier in section 3.5, referred to as Network Similarity.
Secondly, models will be compared by their prediction accuracy when applying various data augmentations to the training dataset.

4.3.1 Network Similarity

Network Similarity calculates a metric to quantify the similarity between two networks. This computation involves determining euclidean distances between each pair of kernels in two neural networks, with the detailed equation outlined in equation 3.1.

For the following tables, Network Similarity was computed by comparing the base model without data augmentations to a neural network with one or two data augmentations applied to its training set. The computation was performed in the same way for the DSADS dataset and the FordA dataset. Additionally, measurements such as mean, std and the minimum and maximum value were computed summarizing the single values from each kernel computation.

Augmentation	Network Similarity (Sum)	Mean	Std. Deviation	Max	Min
Jitter	49.3793	0.0058	0.0031	**0.0279**	**0.0004**
Scaling	**40.4821**	**0.0047**	**0.0025**	0.0327	**0.0004**
Permutation	65.1703	0.0076	0.0039	*0.0383*	0.0005
Magnitude Warp	52.7451	0.0062	0.0032	0.0322	**0.0004**
Slicing	65.6832	0.0077	0.0038	0.0345	*0.0007*
Jitter + Permutation	60.5804	0.0071	0.0035	0.0356	0.0005
Jitter + Slicing	*72.0019*	*0.0084*	*0.0043*	0.0376	0.0006

Table 4.2: Network Similarity in comparison with the base model with no applied data augmentations. Networks were trained with a subset of the DSADS dataset with one or two data augmentations applied. A lower value in the column 'Network Similarity (Sum)' indicates a higher a similarity to the base network. Additional information like the mean are provided as additional columns by saving each distance while computing the metric.

Table 4.2 shows all the computed values for the DSADS dataset. Among all data augmentations, scaling and slicing exhibit the lowest and highest values sum values, respectively. As clarified in section 3.5, a lower sum of distances corresponds to a higher actual Network Similarity. Therefore, in this case, scaling appears to be the closest to the original neural network. Permutation and slicing, on the other hand, both have high sums of euclidean distances, indicating less similarity to the base

network. Jitter and magnitude warp fall in between these extremes.
The mean, standard deviation and minimum and maximum value seem to be closely correlated with the sum of distances. Thus, a higher sum of distances aligns with higher values in these other metrics.
However, there are some outliers. While jitter does not have the lowest sum of distances, it does have the lowest maximum value by a significant margin. On the other hand, permutation has the highest effect on a single kernel compared to all other data augmentations.
Furthermore, table 4.2 includes two combined data augmentations. Interestingly, applying a combination of jitter and permutation results in a higher Network Similarity than only applying permutation to the training dataset. Jitter and slicing produce a significantly higher sum of distances than jitter or slicing itself. However, the combined value is much smaller than the sum of the values for jitter and slicing.

Augmentation	Network Similarity (Sum)	Mean	Std. Deviation	Max	Min
Jitter	3.2865	0.0062	0.0033	0.0253	0.0009
Scaling	**2.4287**	**0.0046**	**0.0025**	**0.0146**	**0.0005**
Permutation	5.8420	0.0111	0.0056	0.0448	*0.0018*
Magnitude Warp	4.8169	0.0091	0.0049	0.0351	0.0011
Slicing	*7.8061*	*0.0148*	*0.0070*	*0.0478*	0.0010
Jitter + Permutation	6.1154	0.0116	0.0054	0.0423	0.0017
Jitter + Slicing	7.2178	0.0137	0.0064	0.0431	0.0013

Table 4.3: Network Similarity in comparison with the base model with no applied data augmentations. Networks were trained with a subset of the FordA dataset with one or two data augmentations applied. Additional information like the mean are provided as additional columns by saving each distance while computing the metric.

Table 4.3 lists values similar to the previous table, using the FordA dataset this time. It is important to note, that the model architecture for these neural networks contains significantly fewer kernels compared to the previous one, resulting in much smaller values in absolute terms.
Similar to the previous results, scaling has the lowest sum of distances and is therefore most similar to the base model. However, while slicing is still at the top when observing the sum, permutation has a significantly lower value than slicing. Despite this, permutation still exhibits a high value when compared to the other augmentations. Jitter and magnitude warp share similar positions as earlier, falling in between the other values.
Interestingly, certain occurrences observed earlier cannot be seen here. Jitter does not have the lowest maximum value. The data augmentation with the lowest sum of distances, scaling, also has the lowest maximum value. Additionally, permutation still exhibits a relatively high maximum value for its Network Similarity.
While the combination of jitter and permutation has a higher sum of distances than the values of jitter and permutation combined by a small margin, its maximum value is still lower than the maximum value of permutation. Jitter and slicing, on the other hand, do not exceed the value of

slicing alone, which does not align with the findings for the DSADS dataset.

Models with slicing and permutation applied to their training dataset likely exhibit a lower similarity to the base models, because slicing and permutation affect the time domain, leading to higher more significant changes induced in the network. For instance, permutation disrupts the temporal structure, potentially destroying patterns and dependencies that might exist over time.
On the other hand, augmentations that target the magnitude domain do not seem to cause the highest changes in networks. This is likely due to an already high variance in the magnitude of the data, resulting in the augmentation not introducing significant alterations. Observing dataset projections in section 4.2.3, might also indicate this effect, as magnitude-transforming augmentations seem to cause lower deviations in generated samples from their original counterparts.

4.3.2 Model performance

Augmentation	100	300	500	700	900	1100	1300	1500
Base	52.3%	72.8%	75.2%	74.8%	76.0%	75.5%	78.8%	79.1%
Jitter	+0.3%	+1.7%	+0.6%	+0.6%	+2.1%	+0.3%	+0.1%	+2.0%
Magnitude Warp	+1.1%	+0.8%	+2.0%	-0.5%	+1.2%	+0.5%	+1.0%	+1.2%
Scaling	+0.3%	+1.2%	+0.4%	+0.1%	+0.2%	+0.4%	-0.9%	+1.4%
Permutation	+2.5%	+1.7%	+0.5%	-0.3%	-0.3%	+0.2%	-0.4%	+0.6%
Slicing	+3.1%	+1.1%	+0.5%	-0.6%	-1.3%	-0.6%	-1.4%	-0.3%
Jitter + Permutation	+2.4%	+2.5%	+1.0%	+1.5%	+1.3%	+0.1%	+0.0%	+1.8%
Jitter + Slicing	+3.3%	+3.2%	+2.5%	+1.0%	+0.9%	+0.2%	-0.5%	+0.8%

Table 4.4: Accuracy for various training dataset sizes for CNNs trained with different data augmentations. Each model was trained on a subset of the DSADS dataset. The columns represent different subset sizes for the training dataset. Each cell in the base row represents the accuracy for the base model for a specific subset size. All other cells represent the change in accuracy compared to the base model for a certain data augmentation.

Model performance was computed according to section 4.1. The table shows per row the applied data augmentations and per column the size of the subset used for training. Each cell in the base row in both tables 4.4 and 4.5 represents the accuracy for the models with no data augmentation at a given subset size. All other cells represent the change in prediction accuracy for the test dataset when applying a specific data augmentation to the dataset. Each combination was trained 10 times and the averaged value of their test accuracy was calculated.

In the following, the results for model performance will be examined for different subset sizes. The chosen subset sizes are 300, 900 and 1500. 300 was selected as its the subset size used for all trained neural networks utilized here. Starting with 300, 900 and 1500 are equally spaced and

were therefore chosen to enable a broad examination.

Upon examining table 4.4 containing data for the DSADS dataset, it is evident that all data augmentations increased prediction accuracy, ranging from 0.8% up to 3.2% for the sample size of 300. The most substantial performance boost was observed when applying both jitter and slicing to the dataset, while the smallest increase was caused by magnitude warp.
For a sample size of 900, not all data augmentations continued to improve accuracy. Permutation and slicing exhibited lower accuracy, with slicing dropping the accuracy by around 1.3%. Other augmentations performed better, but the overall performance increase was lower than with a sample size of 300. In this case, jitter emerged as the best performing data augmentation with an improvement of 2.1%.
Examining the effects of data augmentations for a sample size of 1500 reveals a similar pattern. While the performance improvement seems to decrease further, slicing again leads to a decrease in prediction accuracy. Jitter, once again, is the top performing data augmentation with an accuracy improvement of 2%.

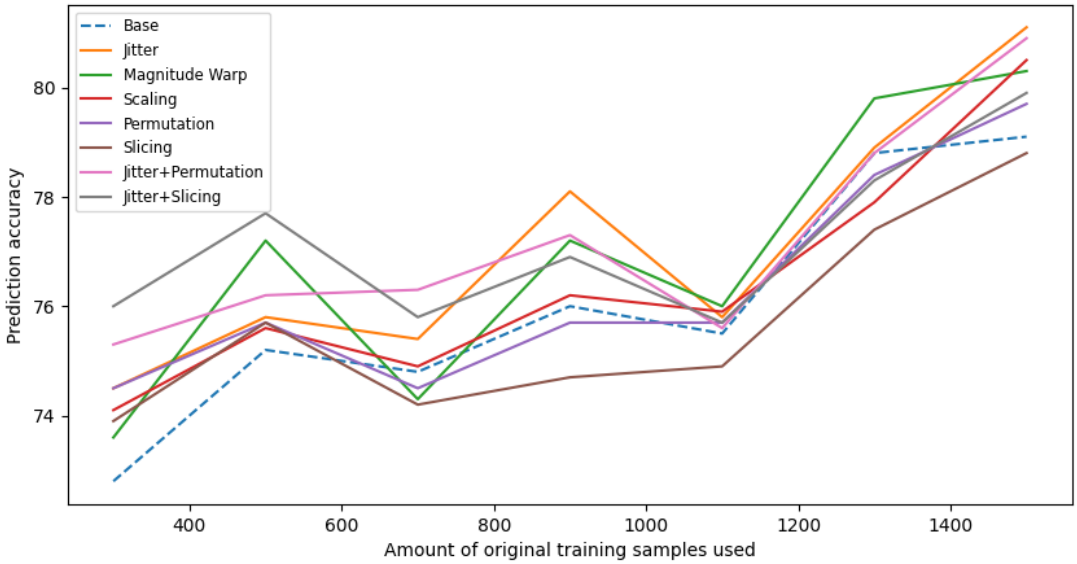


Figure 4.8: Line plot visualization of prediction accuracy over multiple training set sizes for the DSADS dataset. The base model is presented by a blue dashed line to make it easier to compare the other values to. The sample size of 100 is not shown for visibility reasons.

As a general observation, the performance improvement of data augmentations appears to decay with the number of samples in the original dataset. This effect can be observed in figure 4.8, as the values near the end of the x axis tend to closer to the base value than at the beginning. When a substantial portion of the input space is already covered by original data samples, augmenting those may not be as effective. Additionally, applying slicing to larger dataset sizes seems to decrease the prediction performance across multiple subset sizes. An additional noteworthy mention is the combination

of jitter and permutation, which consistently performed close to the top for every examined subset size. This suggests that the synergy between those two augmentations contributes positively to the prediction performance across different dataset sizes.

Augmentation	100	300	500	700	900	1100	1300	1500
Base	75.0%	85.9%	88.4%	90.4%	90.8%	91.2%	91.0%	91.5%
Jitter	+0.2%	+0.0%	+0.1%	-0.2%	-0.1%	-0.2%	+0.1%	-0.2%
Magnitude Warp	+0.2%	+0.5%	+0.2%	-0.1%	+0.2%	-0.1%	+0.3%	-0.1%
Scaling	+0.6%	+0.2%	+0.1%	+0.0%	-0.1%	-0.1%	+0.2%	+0.0%
Permutation	+1.5%	+1.9%	+0.9%	+0.7%	+0.1%	+0.6%	+0.9%	+0.5%
Slicing	+1.5%	-0.4%	-0.1%	-0.9%	-0.9%	-0.5%	-0.7%	-1.0%
Jitter + Permutation	+1.3%	+1.7%	+1.0%	+0.8%	+0.2%	+0.4%	+0.9%	+0.4%
Jitter + Slicing	+1.6%	-0.4%	-0.1%	-0.7%	-0.7%	-0.4%	-0.6%	-0.8%

Table 4.5: Accuracy for various training dataset sizes for CNNs trained with different data augmentations. Each model was trained on a subset of the FordA dataset. The columns represent different subset sizes for the training dataset. 100 in this case means that the original training dataset before augmentation contained 100 samples. Each cell in the base row represents the accuracy for the base model for a specific subset size. All other cells represent the change in accuracy compared to the base model for a certain data augmentation.

Examining table 4.5, the effects are noticeably different for the FordA dataset. Starting at a subset size of 300 as the amount of the original training data samples, data augmentations like jitter, magnitude warp and scaling appear to have little to no effect on the prediction performance. Slicing and the combination of jitter and slicing appear to have a slight negative effect. However, permutation and its combination with jitter increased the prediction performance by 1.9% and 1.7%, respectively. Considering a subset size of 900, similar to the previous subset size, slicing and its combinations show a negative effect, and the positive effect of permutation appears to have decreased. Other data augmentation seem to have no noticeable effect.
Subset size 1500 offers no new insights compared to the previous subset sizes. Slicing and jitter combined with slicing decrease the performance by 1.0% and 0.8% percent, respectively. Permutation and its combination continue to show a small positive effect for this subset size.

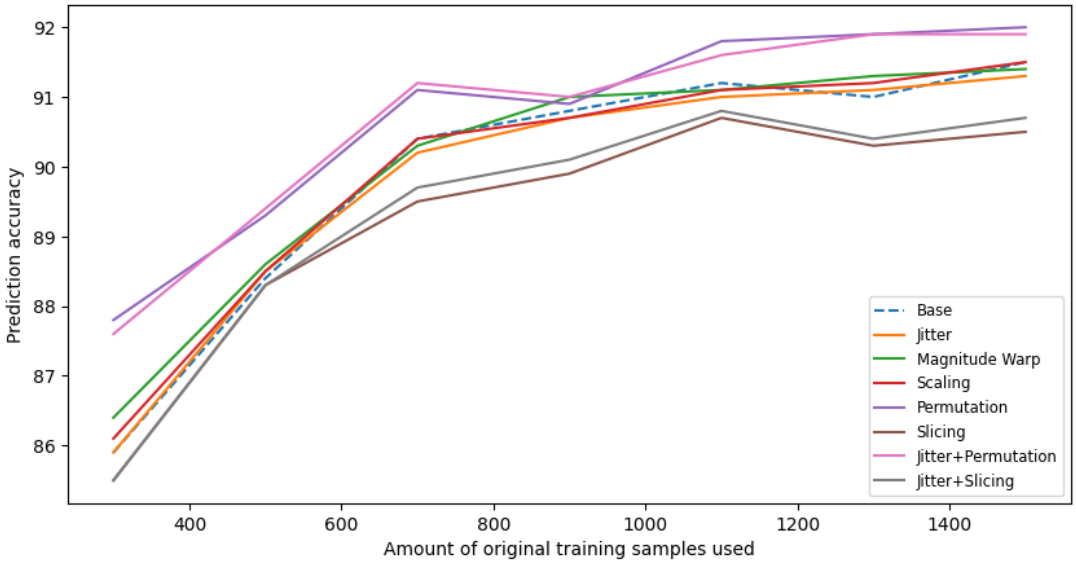


Figure 4.9: Line plot visualization of prediction accuracy over multiple training set sizes for the FordA dataset. The base model is presented by a blue dashed line to make it easier to compare the other values to. The sample size of 100 is not shown for visibility reasons.

To view both the decay of data augmentation effectiveness and the change in performance for various data augmentations at the same time, figure 4.9 can be examined.
Generally, data augmentations that perform random-based transformations in the magnitude domains, such as jitter, magnitude warp and scaling, do not seem to have a significant effect on the prediction performance for the FordA dataset. More variation in the magnitude domain caused by those data augmentations seems neither helpful or harmful. As the FordA dataset consists of engine noises, there might already be a lot of variation in the noise levels.
On the other hand, permutation and therefore also jitter combined with permutation, seemed to lead to performance increase across all subset levels.
As already observed in the data for the DSADS dataset, slicing appears to have a negative effect on prediction performance for all subset sizes above 100. However, contrary to the previous data, the combination of jitter and slicing also have a negative effect. This might be caused by the neutral effect of jitter for this specific dataset not being able to mitigate the negative effects of slicing.

4.4 Combination of Research Findings

Combining the results from different sections explored earlier can deliver some additional insights into the impact of data augmentations on the models latent representation and performance.

The initial observation indicates a correlation between the deviations in dataset projections performed in section 4.2.3 and Network Similarity from section 3.5. Regarding the DSADS dataset, jittering the training data resulted in the smallest maximal value for Network Similarity. Examining the dataset projection for jitter, it was noted that the augmentation caused a minimal spread for the generated samples, almost non-existent near the edges of the visualization. A similar pattern was observed for the projection of scaling, which has the highest similarity to the base network.
Another example is permutation, which had the largest euclidean distance for a specific kernel. The dataset projection for permutation exhibits a significant spread near the edges and generally covered a larger portion of the input space.
In summary, a lower spread in dataset projections seems to be an indication for a higher similarity to the base network after training the model. Consequently, the similarity of the network trained with augmented data seems to be predictable to some extent by examining the deviation of generated samples from the original ones when projecting into 2D space.

Another observation is that even general data augmentations, applicable to any kind of data, may not necessarily be optimal for any dataset. Data augmentations that transform the magnitude domain, such as jitter, scaling and magnitude warp, performed well for the sensor-based DSADS dataset. However, when evaluating the changes in model performance for these data augmentations in the FordA dataset, it appears that they did not have a significant impact on prediction performance. As mentioned earlier in section 4.3.2, this could be caused by an already high variance in the magnitude of the training data.
Another example is the application of a combination of jitter and slicing, which provided a decent improvement in performance for the DSADS dataset, but caused a lower prediction accuracy when applied to the FordA dataset.
Given these observations, the selection of data augmentations should be adapted to the specific characteristics of the dataset.
To assess the potential effectiveness of specific data augmentation technique for a given dataset, it is advised to utilize a combination of diverse approaches. For instance, in the context of slicing, methods such as Network Similarity might initially let slicing appear promising due to its substantial impact on the neural network. Nevertheless, a comprehensive evaluation considering both performance metrics and dataset projections reveals that the diminished performance could be attributed to the generation of overly similar samples.

The analysis of the Activation Atlas of the augmentation created by combining jitter and permutation suggests that the combination of data augmentations produces effects not observed in each of the augmentation alone. Jitter appeared to introduce high-frequency components in the maximized activations signals, while permutation seemed induce smoothing. However, when combining both augmentations, it appears that a portion of the introduction of high-frequency components by jitter seems to be mitigated to some extent by the perceived smoothing effect of permutation.
A similar effect becomes apparent when examining the network similarity for the combination of these data augmentations. While permutation alone has one of the lowest similarities compared to the base model, jitter appears to counteract the effects of permutation on the network, resulting in an overall higher similarity to the base network.

For both datasets, using slicing as a data augmentation seemed to reduce model performance across different sample sizes, as observed in section 4.3.2. An indication of this diminished performance may be found in the dataset projection shown in section 4.2.3. Slicing appears to result in a higher spread when comparing generated samples to original samples. While the spread itself might not be concerning, the fact that some of the generated samples from different classes overlap raises potential issues. These overlapped samples may cause unfavorable weight updates in the neural networks, leading to incorrect predictions. Therefore, the utilization of slicing should be approached with caution and thoroughly evaluated.

5 Conclusion

5.1 Methodological Limitations and Future Research Directions

One of the main assumptions in this thesis is that a kernel in the same convolutional layer, at the same position, extracts the same feature for two different models. The steps to ensure the truth of this assumption are further explained in section 3.4. Without guaranteed reproducibility, many of the techniques used here would not work as expected when performing a comparison of two neural networks. However, when only examining a single CNN, these assumptions are not as critical.

Another limitation of this thesis is the inability to fully explore all parts of a neural network due to the limited amount of time and space available. Neural networks, especially CNNs, consist of many interacting parts, making it challenging to fully comprehend even for a single network. Comparing various models trained on different data augmentations and datasets is therefore a complex task. However, the techniques proposed in this thesis offer a helpful approach to tackel these challenges.

As for future research, exploring methods to visualize an entire CNN instead of distinct convolutional layers could provide valuable insights into the workings of deep neural networks.

The combination of different data augmentations, especially jitter and permutation, demonstrated promising performance improvements and interesting interactions. Further exploration of these combinations for time series data may be beneficial to uncover additional insights.

5.2 Summary

This thesis investigated the influence of data augmentations on the latent representation and performance of convolutional neural networks (CNNs) for time series data. In the pursuit of comprehending large neural networks, often considered as black boxes due to the interactions among numerous non-linear components, an interactive web application was developed to facilitate exploration. Prior to utilizing the web application, multiple models were trained with different data augmentations applied to their training datasets. The training datasets of these models consisted of

different datasets, both univariate and multivariate time series.
To explore the features extracted by a specific convolutional layer, activation maximization (AM) was employed in conjunction with dimensionality reduction techniques to generate a map of all visual features, referred to as Activation Atlas. While the web application incorporates additional interactive visualizations to enhance the understanding of deep neural network functionality, a quantitative approach was also employed. This involved examining the models performance and network similarity, a metric introduced in this thesis.
Considering the significant challenge of understanding neural networks, particularly due to their depth, this thesis marks a meaningful step toward unraveling the complexities of these intricate systems.

Bibliography

[1] Guozhong An. “The Effects of Adding Noise During Backpropagation Training on a Generalization Performance”. In: *Neural Computation* 8.3 (1996), pp. 643–674. ISSN: 0899-7667. DOI: 10.1162/neco.1996.8.3.643.

[2] antoinebrl. *Convolution Sandbox*. https://antoinebrl.github.io/blog/conv1d/. Accessed: 2024-01-01.

[3] Billur Barshan and Kerem Altun. *Daily and Sports Activities*. UCI Machine Learning Repository. DOI: https://doi.org/10.24432/C5C59F. 2013.

[4] Estela Bee Dagum and Silvia Bianconcini. *Seasonal Adjustment Methods and Real Time Trend-Cycle Estimation*. Statistics for Social and Behavioral Sciences. Cham: Springer International Publishing, 2016. ISBN: 978-3-319-31820-2. DOI: 10.1007/978-3-319-31822-6. URL: http://link.springer.com/10.1007/978-3-319-31822-6.

[5] Konstantinos Benidis et al. “Deep Learning for Time Series Forecasting: Tutorial and Literature Survey”. In: *ACM Comput. Surv.* 55.6 (2022). ISSN: 0360-0300. DOI: 10.1145/3533382. URL: https://doi.org/10.1145/3533382.

[6] Christoph Bergmeir, Rob J. Hyndman, and José M. Benítez. “Bagging exponential smoothing methods using STL decomposition and Box–Cox transformation”. In: *International Journal of Forecasting* 32.2 (Apr. 2016), pp. 303–312. ISSN: 0169-2070. DOI: 10.1016/j.ijforecast.2015.07.002.

[7] John S. Bridle. “Probabilistic Interpretation of Feedforward Classification Network Outputs, with Relationships to Statistical Pattern Recognition”. en. In: *Neurocomputing*. Ed. by Françoise Fogelman Soulié and Jeanny Hérault. NATO ASI Series. Berlin, Heidelberg: Springer, 1990, pp. 227–236. ISBN: 978-3-642-76153-9. DOI: 10.1007/978-3-642-76153-9_28.

[8] Shan Carter et al. “Activation Atlas”. In: *Distill* (2019). https://distill.pub/2019/activation-atlas. DOI: 10.23915/distill.00015.

[9] N. V. Chawla et al. “SMOTE: Synthetic Minority Over-sampling Technique”. en. In: *Journal Of Artificial Intelligence Research* (2002). DOI: 10.1613/jair.953. URL: https://arxiv.org/abs/1106.1813v1.

[10] S. Chopra, R. Hadsell, and Y. LeCun. “Learning a similarity metric discriminatively, with application to face verification”. In: *2005 IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR'05)*. Vol. 1. June 2005, 539–546 vol. 1. DOI: 10.1109/CVPR.2005.202. URL: https://ieeexplore.ieee.org/document/1467314.

[11] Robert B. Cleveland et al. “STL: A Seasonal-Trend Decomposition Procedure on Loess”. In: *Journal of Official Statistics* 6.1 (1990), pp. 3–73.

[12] Hoang Anh Dau et al. *The UCR Time Series Classification Archive*. https://www.cs.ucr.edu/~eamonn/time_series_data_2018/. 2018.

[13] Jia Deng et al. “ImageNet: A large-scale hierarchical image database”. In: *2009 IEEE Conference on Computer Vision and Pattern Recognition*. 2009, pp. 248–255. DOI: 10.1109/CVPR.2009.5206848.

[14] Víctor Gómez and Agustín Maravall Herrero. *Seasonal adjustment and signal extraction in economic times series*. en. Documento de trabajo / Banco de España, Servicio de Estudios. Madrid: Banco de España, 1998. ISBN: 978-84-7793-598-8.

[15] Ian Goodfellow et al. “Generative Adversarial Nets”. en. In: ().

[16] R. Hadsell, S. Chopra, and Y. LeCun. “Dimensionality Reduction by Learning an Invariant Mapping”. In: *2006 IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR'06)*. Vol. 2. June 2006, pp. 1735–1742. DOI: 10.1109/CVPR.2006.100. URL: https://ieeexplore.ieee.org/document/1640964.

[17] Alon Halevy, Peter Norvig, and Fernando Pereira. “The Unreasonable Effectiveness of Data”. In: *IEEE Intelligent Systems* 24.2 (Mar. 2009), pp. 8–12. ISSN: 1941-1294. DOI: 10.1109/MIS.2009.36.

[18] Kaiming He et al. “Deep Residual Learning for Image Recognition”. In: *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)* (2016), pp. 770–778. DOI: 10.1109/CVPR.2016.90.

[19] Elad Hoffer and Nir Ailon. “Deep Metric Learning Using Triplet Network”. en. In: *Similarity-Based Pattern Recognition*. Ed. by Aasa Feragen, Marcello Pelillo, and Marco Loog. Lecture Notes in Computer Science. Cham: Springer International Publishing, 2015, pp. 84–92. ISBN: 978-3-319-24261-3. DOI: 10.1007/978-3-319-24261-3_7.

[20] Rob J. Hyndman and George Athanasopoulos. *Forecasting: Principles and practice*. OTexts, 2018.

[21] Sergey Ioffe and Christian Szegedy. “Batch normalization: accelerating deep network training by reducing internal covariate shift”. In: *Proceedings of the 32nd International Conference on International Conference on Machine Learning - Volume 37*. ICML'15. Lille, France: JMLR.org, 2015, pp. 448–456.

[22] Hassan Ismail Fawaz et al. “Deep Learning for Time Series Classification: A Review”. In: *Data Min. Knowl. Discov.* 33.4 (2019), pp. 917–963. ISSN: 1384-5810. DOI: 10.1007/s10618-019-00619-1. URL: https://doi.org/10.1007/s10618-019-00619-1.

[23] Brian Kenji Iwana and Seiichi Uchida. “An empirical survey of data augmentation for time series classification with neural networks”. In: *PLOS ONE* 16.7 (July 2021), pp. 1–32. DOI: 10.1371/journal.pone.0254841. URL: https://doi.org/10.1371/journal.pone.0254841.

[24] N. Jaitly and Geoffrey E. Hinton. “Vocal Tract Length Perturbation (VTLP) improves speech recognition”. In: 2013. URL: https://www.semanticscholar.org/paper/Vocal-Tract-Length-Perturbation-(VTLP)-improves-Jaitly-Hinton/f79174a79b0391b6

[25] Diederik P. Kingma and Jimmy Ba. “Adam: A Method for Stochastic Optimization”. In: arXiv:1412.6980 (Jan. 2017). arXiv:1412.6980 [cs]. DOI: 10.48550/arXiv.1412.6980. URL: http://arxiv.org/abs/1412.6980.

[26] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. “ImageNet Classification with Deep Convolutional Neural Networks”. In: *Advances in Neural Information Processing Systems*. Ed. by F. Pereira et al. Vol. 25. Curran Associates, Inc., 2012. URL: https://proceedings.neurips.cc/paper_files/paper/2012/file/c399862d3b9d6b76c8436e924a68c45b-Paper.pdf.

[27] Y. Lecun et al. “Gradient-based learning applied to document recognition”. In: *Proceedings of the IEEE* 86.11 (1998), pp. 2278–2324. DOI: 10.1109/5.726791.

[28] Aitor Lewkowycz and Guy Gur-Ari. “On the training dynamics of deep networks with L2 regularization”. In: *Advances in Neural Information Processing Systems*. Vol. 33. Curran Associates, Inc., 2020, pp. 4790–4799. URL: https://proceedings.neurips.cc/paper/2020/hash/32fcc8cfe1fa4c77b5c58dafd36d1a98-Abstract.html.

[29] Qianli Ma et al. “A Survey on Time-Series Pre-Trained Models”. In: arXiv:2305.10716 (2023). arXiv:2305.10716 [cs]. DOI: 10.48550/arXiv.2305.10716. URL: http://arxiv.org/abs/2305.10716.

[30] Laurens van der Maaten and Geoffrey Hinton. “Visualizing Data using t-SNE”. In: *Journal of Machine Learning Research* 9.86 (2008), pp. 2579–2605. ISSN: 1533-7928.

[31] J. MacQueen. “Some methods for classification and analysis of multivariate observations”. In: *Proceedings of the Fifth Berkeley Symposium on Mathematical Statistics and Probability, Volume 1: Statistics*. Vol. 5.1. University of California Press, Jan. 1967, pp. 281–298. URL: https://projecteuclid.org/ebooks/berkeley-symposium-on-mathematical-statistics-and-probability/Proceedings-of-the-Fifth-Berkeley-Symposium-on-Mathematical-Statistics-and/chapter/Some-methods-for-classification-and-analysis-of-multivariate-observations/bsmsp/1200512992.

[32] Leland McInnes et al. “UMAP: Uniform Manifold Approximation and Projection”. en. In: *Journal of Open Source Software* 3.29 (Sept. 2018), p. 861. ISSN: 2475-9066. DOI: 10.21105/joss.00861.

[33] Vinod Nair and Geoffrey E. Hinton. “Rectified Linear Units Improve Restricted Boltzmann Machines”. In: *Proceedings of the 27th International Conference on International Conference on Machine Learning*. ICML’10. Haifa, Israel: Omnipress, 2010, pp. 807–814. ISBN: 9781605589077.

[34] Anh Nguyen, Jason Yosinski, and Jeff Clune. “Deep Neural Networks Are Easily Fooled: High Confidence Predictions for Unrecognizable Images”. In: 2015, pp. 427–436. URL: https://www.cv-foundation.org/openaccess/content_cvpr_2015/html/Nguyen_Deep_Neural_Networks_2015_CVPR_paper.html.

[35] Anh Nguyen et al. “Synthesizing the preferred inputs for neurons in neural networks via deep generator networks”. In: *Advances in Neural Information Processing Systems*. Vol. 29. Curran Associates, Inc., 2016. URL: https://proceedings.neurips.cc/paper/2016/hash/5d79099fcdf499f12b79770834c0164a-Abstract.html.

[36] Johannes Pöppelbaum, Gavneet Singh Chadha, and Andreas Schwung. “Contrastive Learning Based Self-Supervised Time-Series Analysis”. In: *Appl. Soft Comput.* 117.C (2022). ISSN: 1568-4946. DOI: 10.1016/j.asoc.2021.108397. URL: https://doi.org/10.1016/j.asoc.2021.108397.

[37] Zhuwei Qin et al. “How convolutional neural networks see the world — a survey of Convolutional Neural Network visualization methods”. In: *Mathematical Foundations of Computing* 1.2 (2018), pp. 149–180. DOI: 10.3934/mfc.2018008.

[38] Ramprasaath R. Selvaraju et al. “Grad-CAM: Visual Explanations from Deep Networks via Gradient-based Localization”. In: *International Journal of Computer Vision* 128.2 (Feb. 2020). arXiv:1610.02391 [cs], pp. 336–359. ISSN: 0920-5691, 1573-1405. DOI: 10.1007/s11263-019-01228-7.

[39] Artemios-Anargyros Semenoglou, Evangelos Spiliotis, and Vassilis Assimakopoulos. “Data augmentation for univariate time series forecasting with neural networks”. In: *Pattern Recognition* 134 (Feb. 2023), p. 109132. DOI: 10.1016/j.patcog.2022.109132.

[40] Connor Shorten and Taghi M. Khoshgoftaar. “A survey on Image Data Augmentation for Deep Learning”. In: *Journal of Big Data* 6.1 (2019), p. 60. ISSN: 2196-1115. DOI: 10.1186/s40537-019-0197-0.

[41] Karen Simonyan, Andrea Vedaldi, and Andrew Zisserman. “Deep Inside Convolutional Networks: Visualising Image Classification Models and Saliency Maps”. In: arXiv:1312.6034 (Apr. 2014). arXiv:1312.6034 [cs]. DOI: 10.48550/arXiv.1312.6034. URL: http://arxiv.org/abs/1312.6034.

[42] Nitish Srivastava et al. “Dropout: A Simple Way to Prevent Neural Networks from Overfitting”. In: *Journal of Machine Learning Research* 15.56 (2014), pp. 1929–1958. ISSN: 1533-7928.

[43] Richard Szeliski. *Computer vision: Algorithms and applications*. Springer Nature, 2022.

[44] Antonio Torralba, Rob Fergus, and William T. Freeman. “80 Million Tiny Images: A Large Data Set for Nonparametric Object and Scene Recognition”. In: *IEEE Transactions on Pattern Analysis and Machine Intelligence* 30.11 (2008), pp. 1958–1970. ISSN: 1939-3539. DOI: 10.1109/TPAMI.2008.128.

[45] Xiaozhe Wang, Kate Smith, and Rob Hyndman. “Characteristic-Based Clustering for Time Series Data”. en. In: *Data Mining and Knowledge Discovery* 13.3 (2006), pp. 335–364. ISSN: 1573-756X. DOI: 10.1007/s10618-005-0039-x.

[46] Zhiguang Wang, Weizhong Yan, and Tim Oates. “Time series classification from scratch with deep neural networks: A strong baseline”. In: *2017 International Joint Conference on Neural Networks (IJCNN)*. May 2017, pp. 1578–1585. DOI: 10.1109/IJCNN.2017.7966039. URL: https://ieeexplore.ieee.org/document/7966039/footnotes#footnotes.

[47] Naoya Yoshimura, Takuya Maekawa, and Takahiro Hara. “Toward Understanding Acceleration-based Activity Recognition Neural Networks with Activation Maximization”. In: *2021 International Joint Conference on Neural Networks (IJCNN)*. July 2021, pp. 1–8. DOI: 10.1109/IJCNN52387.2021.9533888. URL: https://ieeexplore.ieee.org/abstract/document/9533888.

[48] Jason Yosinski et al. “Understanding Neural Networks Through Deep Visualization”. In: arXiv:1506.06579 (June 2015). arXiv:1506.06579 [cs]. DOI: 10.48550/arXiv.1506.06579. URL: http://arxiv.org/abs/1506.06579.

[49] Matthew D. Zeiler and Rob Fergus. “Visualizing and Understanding Convolutional Networks”. en. In: *Computer Vision – ECCV 2014*. Ed. by David Fleet et al. Lecture Notes in Computer Science. Cham: Springer International Publishing, 2014, pp. 818–833. ISBN: 978-3-319-10590-1. DOI: 10.1007/978-3-319-10590-1_53.

[50] Bendong Zhao et al. “Convolutional neural networks for time series classification”. In: *Journal of Systems Engineering and Electronics* 28.1 (2017), pp. 162–169. DOI: 10.21629/JSEE.2017.01.18.

[51] Bolei Zhou et al. “Learning Deep Features for Discriminative Localization”. In: *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. June 2016, pp. 2921–2929. DOI: 10.1109/CVPR.2016.319. URL: https://ieeexplore.ieee.org/document/7780688.