Anna Sophie Nymoen Tveit

Leveraging a Convolutional Neural Network for Real-Time Classification of Distributed Acoustic Sensing Data alongside a Railway

Master's Thesis in Electronic Systems Design and Innovation
Supervisor: Hefeng Dong
Co-supervisor: Kevin Growe
March 2024

Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Department of Electronic Systems

NTNU

ABSTRACT

The aim of this research was to create a framework for a live-monitoring system of railways using distributed acoustic sensing data and a convolutional neural network. To achieve the goal, this work exploited a total of 70 minutes of distributed acoustic sensing data from a 51 km long railway line section between Trondheim and Støren (Trøndelag, Norway). A common pre-processing flow was followed by the computation of rolling RMS windows of 60 seconds and 1.5 kilometer with 50 % overlap in both time and space. The resulting data windows were normalized and fed to a convolutional neural network for classification of 12 separate signal classes, such as trains, cars, various noise types and unknown events. A dataset of 5,000 images per class was acquired by the use of manual labeling and various augmentation techniques, resulting in a full dataset of 60,000 images. Extensive hyperparameter tests were conducted to increase the performance of the network. The training of the convolutional neural network resulted in a general testing accuracy of 84.98 %, with specific class accuracies ranging from 70.65 % to 98.13 %. Additionally, live-classification of the final model was tested by applying a forward pass of unseen data through the model every 30 seconds for each 1.5 kilometer long overlapping segment along the entire 51 kilometer long railway line section.

ABSTRAKT

Formålet med dette forskningsprosjektet var å utvikle et rammeverk for et sanntidsovervåkningssystem for jernbaner, ved bruk av "distributed acoustic sensing" data og et konvolusjonelt nevralt nettverk. For å nå dette målet, ble totalt 70 minutter "distributed acoustic sensing" data fra et 51 km langt jernbanelinjesegment mellom Trondheim og Støren (Trøndelag, Norge) bearbeidet. En kjent preprosesseringsteknikk ble brukt, hvor hensikten var å regne ut rullende RMS-vinduer på 60 sekunder og 1.5 kilometer med 50 % overlapp i tid og rom. De resulterende vinduene ble normalisert og matet inn i et konvolusjonelt nevralt nettverk for klassifisering av 12 ulike signalklasser, inkludert tog, biler, ulike støytyper og ukjente hendelser. Et datasett bestående av 5,000 bilder per klasse ble fremstilt via manuell bildemerking (labeling) og ulike bildemanipuleringsteknikker, som resulterte i et totalt datasett bestående av 60,000 bilder. Treningen av nettverket resulterte i en generell testnøyaktighet på 84.98 %, med spesifikke nøyaktigheter per klasse varierende fra 70.65 % til 98.13 %. I tillegg ble sanntidsklassifisering av den endelige modellen simulert ved å sende usett data fremover i nettverket hvert 30. sekund for hvert 1.5 kilometer lange overlappende segment langs hele the 51 kilometer lange jernbanelinjesegmentet.

PREFACE

This research is a result of the course "TFE4930 - Electronic Systems Design, Master's Thesis" as the final part of a 2-year Master of Science in Engineering programme at the Norwegian University of Science and Technology (NTNU).

I am so grateful for having had the opportunity to delve into subjects like artificial intelligence, distributed acoustic sensing and big data processing during this research period. I would like to direct my gratitude towards my supervisor Hefeng Dong for helpful guidance on and support through the research period. And to my co-supervisor, Kevin Growe, for being an excellent collaborator in this research. So thank you both, for allowing me to be part of your team during this research period. And thank you Robin, for your endless support. Lastly, I acknowledge Bane NOR and Alcatel Submarine Networks for conducting the data acquisition for this project.

The entire code written for this research is stored as a GitHub Repository and can be viewed upon request.

Trondheim
March 11, 2024
Anna Sophie Nymoen Tveit

CONTENTS

Abstract i

Abstrakt ii

Preface iii

Acronyms vi

1 Introduction 1
1.1 Research Description . . . 3
1.1.1 Outline . . . 3

2 Theory 4
2.1 Distributed Acoustic Sensing . . . 4
2.2 Artificial Intelligence . . . 6
2.2.1 Artificial Neural Networks . . . 6
2.2.2 Convolutional Neural Networks . . . 7
2.2.2.1 Convolutional Layer . . . 7
2.2.3 Activation . . . 8
2.2.3.1 Pooling Layer . . . 9
2.3 Normalization . . . 10
2.4 Model Training and Hyperparameter Tuning . . . 10
2.4.1 Loss Function . . . 12
2.4.2 Backpropagation Algorithm . . . 13
2.4.2.1 Optimization and Gradient Descent . . . 13
2.5 Model Performance Evaluation . . . 14
2.5.1 Learning Curves . . . 14
2.5.2 Confusion Matrices and Precision/Recall Scores . . . 14

3 Methods 16
3.1 Data Acquisition . . . 17
3.2 Data Preprocessing . . . 18
3.3 Data Labeling . . . 19
3.3.1 Labeling Considerations . . . 24
3.4 Data Augmentation . . . 24
3.5 Model Building and Training . . . 26
3.5.1 Base Model Configurations . . . 26
3.5.2 Training . . . 27
3.6 Performance Evaluation and Hyperparameter Tuning . . . 27

3.6.1 Hyperparameter Tuning . . . 27
1. Model Iteration . . . 27
2. Learning Rates and Optimizer Iteration . . . 29
3. Additional Hyperparameters . . . 30
4. Final Run and Live-Classification . . . 30
3.6.2 Performance Evaluation . . . 31

4 Results **32**
1. Model Iteration . . . 33
2. Learning Rates and Optimizer Iteration . . . 34
3. Additional Hyperparameters . . . 35
4. Final Run and Live-Classification . . . 36
4.1 Classification Results and Evaluation . . . 38
4.1.1 Confusion Matrix . . . 38
4.1.2 Manual Inspection of Incorrect Predictions . . . 39

5 Discussion **40**
5.1 Result Interpretations . . . 40
5.1.1 Class Diversity vs. Accuracy . . . 40
5.1.2 Vehicle Classes . . . 41
5.1.3 Incorrect Class Labels in Dataset . . . 41
5.2 Implications . . . 42
5.3 Methods Assessment and Limitations . . . 43
5.3.1 Future Work . . . 44
5.3.1.1 Dataset Development . . . 45
5.3.1.2 Modeling and Training . . . 45

6 Conclusions **46**

References **46**

Appendices **51**
A BasicModelMediumParams Architecture . . . 51
B Training and Validation Loops . . . 53
C Accuracy and Loss Curves for Different Models . . . 55
D Learning Curves with Lower Learning Rate . . . 57
E Additional Incorrect Predictions . . . 58

ACRONYMS

AI Artificial Intelligence

ANN Artificial Neural Network

C-OTDR Coherent Optical Time Domain Reflectometry

CNN Convolutional Neural Network

DAS Distributed Acoustic Sensing

FNN Feedforward Neural Network

IU Interrogator Unit

ReLU Rectified Linear Unit

RMS Root Mean Square

SGD Stochastic Gradient Descent

SNR Signal-to-Noise Ratio

CHAPTER 1

INTRODUCTION

The field of Artificial Intelligence (AI) has seen great advancements in the recent years. The sub-field of computer vision, and in particular deep learning is no exception of that. Concepts such as ChatGPT, self-driving cars and voice assistants like Siri or Alexa are just a few developments that have become very popular in recent times [1][2]. Similarly, Distributed Acoustic Sensing (DAS) has gained popularity and growth for its cost-effective and versatile way of collecting acoustic or ground vibration data with high temporal and spatial resolution [3][4].

The large amounts of data generated during DAS acquisition, often exceed several terabytes per day. Manual processing of those amounts of data is not feasible, thus automated methods are required for the task. A solution can be the use of AI, in particular the sub-fields that specialize in tasks like pattern recognition and categorization. With this in mind, the use of DAS for data collection and AI for data organization and pattern recognition opens up a wide range of applications. There are several relevant articles on this topic. For instance, Kayan et al. used deep learning state-of-the-art methods on DAS data for classification of various DAS signals in 2022 [5]. Peng et al. identified and classified human movement using deep learning on DAS data in 2020 [6]. In 2023, Corera et al. used machine learning methods for identification and classification of vehicles using DAS data [7]. Additionally, Nayak and Ajo-Franklin used DAS methods to successfully detect earthquakes in 2021 [8], while deep learning methods were used on DAS data for the same purpose by Hernández et al. in 2022 [9].

Additionally, previous research indicates that Convolutional Neural Network (CNN)s are well suited tools for processing of DAS data. For instance, Wu et al. used a CNN for pattern recognition on DAS data in 2023 [10]. Rahman et al. used a combination of a CNN and other learning methods for monitoring railway tracks using DAS data in 2024 [11]. Moreover, Wang et al. researched CNN application related to railway tracks using DAS data in 2021 [12].

[1] https://sdgs.un.org/goals

There are several advantages in using DAS for data collection. Among other things, the fiber often coincides with vehicle and railroads, making DAS easily available for data acquisition. Besides, DAS is considerably cheaper than other conventional sensors (e.g. geophones), making it a cost-effective option. The fiber is usually buried, thus the data is not subject to any weather effects. Conversely to other sensors, the fiber itself does not require any power supply, which benefits acquisitions over long distances. DAS is sustainable and environmental friendly, making it a viable contribution to UN's sustainability goals[1]. All these benefits make DAS an attractive candidate for data collection.

There is currently a growing need for effective and accurate monitoring solutions [13]. Problems with animal crossings/collisions and other kinds of disturbance could be prevented with a live-monitoring system along the railway line. One solution could be the utilization of DAS and a CNN. Based on previous research and the inherent properties of both DAS and CNNs, a monitoring system integrating both technologies seems promising. In order to assess the feasibility of such a system, this research aims to create a framework for a live-monitoring system of railway DAS signals, by the use of a CNN.

If successful, this research could provide railway companies with a valuable tool, enabling monitoring of trains, cars, noise and other events along entire railway lines. Moreover, the methods developed in this research may have potential for even further development, with potential applications like traffic analysis, event detection of railway or traffic accidents, animal crossings, and possibly even anomaly detections of hazards like landslides. Additionally, this research may turn out to be a valuable contribution to today's growing pool of research regarding the integration of AI and DAS. Even though the main focus of this thesis is on railway monitoring, due to its adaptability, the developed workflow may be applicable in other research fields and industries.

An additional benefit of this research is its contribution to the United Nations' sustainability goals. A number of the goals are highly relevant to this research, including the development of sustainable monitoring practices and efficient resource management. Other goals, such as improved hazard assessment and facilitation of early structural issue detection may also be relevant. Strategic integration of AI and DAS technology may lead to more effective and environmentally conscious monitoring solutions, which addresses the challenges of infrastructure sustainability, urbanization and climate resilience, defined by goals 9.1[1], 11.3[2] and 13.1[3] respectively.

[1] Develop quality, reliable, sustainable and resilient infrastructure, including regional and transborder infrastructure, to support economic development and human well-being, with a focus on affordable and equitable access for all

[2] By 2030, enhance inclusive and sustainable urbanization and capacity for participatory, integrated and sustainable human settlement planning and management in all countries

[3] Strengthen resilience and adaptive capacity to climate-related hazards and natural disasters in all countries

1.1 Research Description

The main goal of this research is to be able to classify different signal classes on a railway in real-time, after building and training a CNN specialized for the given task. For this goal, the research explores DAS data collected during 70 minutes on a railway in between Marienborg and Støren in Trøndelag, on the 31st of August in 2021. A map of the fiber placement is displayed in Figure 3.2 in the Methods section. This research builds on the previous work by the author's project report, "Leveraging Unsupervised Machine Learning Methods for Distributed Acoustic Sensing" [14], finished in June 2023. Where the earlier work was based upon the research of unsupervised learning, this work is based on supervised learning. In this research, a method of labeling and augmenting training data is developed, before the training data is passed through the specialized CNN for classification. The results of the classification are listed and discussed, and the model is further tested on a dataset unseen by the model as a simulation of live classification. Based on the findings of the research, future work recommendations are given.

1.1.1 Outline

Chapter 2 includes basic theoretic explanations of the principles utilized throughout the research.
Chapter 3 describes a full workflow ranging from data acquisition to the final training of the CNN.
Chapter 4 displays all relevant findings/results from this research.
Chapter 5 contains a discussion which reflects around the project methods, results and its relevance to research today, in addition to suggestions for future work.
Chapter 6 is a research summary.

CHAPTER 2

THEORY

The aim of this chapter is to provide the reader with all the necessary theoretical background needed in order to understand the following contents of the subsequent chapters of this thesis.

2.1 Distributed Acoustic Sensing

There are multiple techniques for performing DAS [15]. One commonly employed technique involves the application of Coherent Optical Time Domain Reflectometry (C-OTDR), which is based on the measurement of Rayleigh scattering [16].

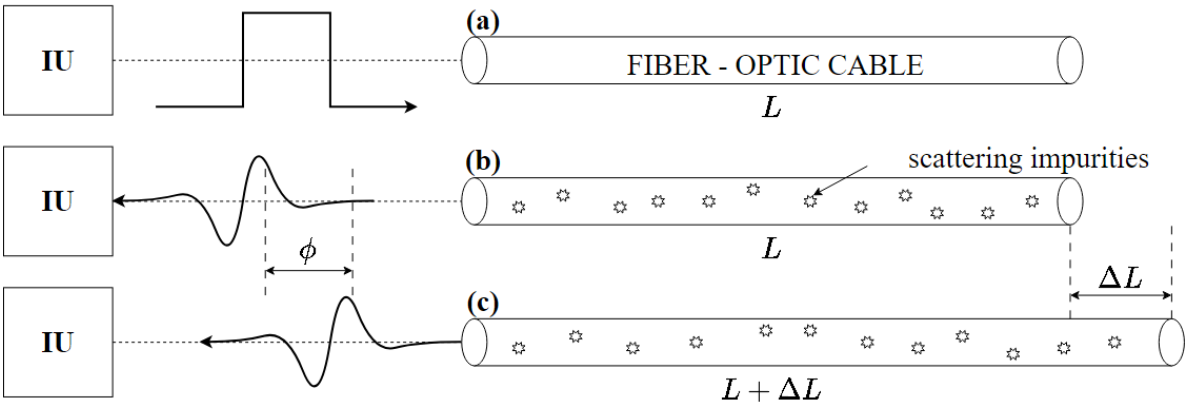

Figure 2.1: Working principle of C-OTDR. In (a) the Interrogator Unit (IU) sends a laser pulse into the fiber. (b) and (c) illustrates the fiber segment as L and the fiber elongation as ΔL. This elongation is causing a phase shift, ϕ, which is measured by the interrogator unit, illustrated to the left of the figure. Figure adapted from Lowrie and Fichtner [17].

Following the C-OTDR principle, an Interrogator Unit (IU) is placed at one end of the fiber-optic cable, sending light pulses into it. Part of the light is reflected at the inherent impurities within the fiber and is recorded by the IU. If a segment of the fiber is either compressed or elongated, the change of position of the impurities between consecutive interrogations results in a phase shift of the back-scattered laser pulse [18]. See Figure 2.1 for an illustration of C-OTDR.

The phase shift measured by the IU has a proportional relationship with the strain acting on the cable [18], and can be calculated using the following set of formulas, adopted from Taweesintananon et al. [19]. If $c \approx 3 * 10^8$ is the speed of light in vacuum, n_g is the refractive index of the fiber and $\Delta\tau$ is the sampling period at the optical receiver, the spatial sampling interval, or SSI, can be calculated using the following formula:

$$SSI = \frac{c}{2n_g}\Delta\tau \tag{2.1}$$

Furthermore, the phase of the back-scattered light, ϕ_x, is calculated in Formula 2.2, where x is the spatial sample location, or channel, and λ_0 is the free space wavelength.

$$\phi_x = \frac{4\pi n_g x}{\lambda_0} \tag{2.2}$$

The IU measures the time-differentiated phase change $\Delta\dot{\phi}$ over a small subsection of the fiber, the so-called gauge length (L_G), as:

$$\Delta\dot{\phi}_x = \dot{\phi}_{avg,x+L_G/2} - \dot{\phi}_{avg,x-L_G/2}, \tag{2.3}$$

where L_G is set by the operator as a multiple of the SSI:

$$L_G = N_{\Delta\tau} * SSI \tag{2.4}$$

Both the Signal-to-Noise Ratio (SNR) and the spatial resolution is dependent on the gauge length. A larger gauge length will cause a higher SNR, while also lowering the spatial resolution. If ζ is the strain-optic coefficient, a material parameter of the fiber, the longitudinal strain-rate can then be computed using the following formula:

$$\dot{\epsilon}_{xx,x} = \frac{\lambda_0}{4\pi n_g \zeta L_G}\Delta\dot{\phi}_x \tag{2.5}$$

Finally, strain is more stable and less susceptible to noise than the strain rate, and is commonly used instead:

$$\epsilon_{xx,x} = \int \dot{\epsilon}_{xx,x} dt \tag{2.6}$$

2.2 Artificial Intelligence

Simply put, AI is the simulation of human intelligence, performed by machines. To realize this, deep learning exists as a technique for the machines to achieve this simulated intelligence [20]. Neural networks, like Artificial Neural Network (ANN)s or CNNs are the foundation of deep learning algorithms.

2.2.1 Artificial Neural Networks

There are two broad types of ANNs: Feedforward Neural Network (FNN)s and recurrent, or bidirectional, neural networks. The FNN is the basis of CNNs, which are commonly used in image classification problems.

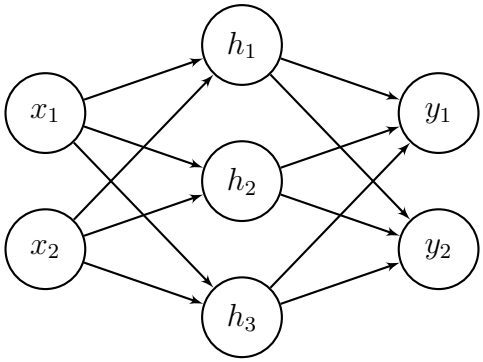

Figure 2.2: A FNN with an input layer consisting of two nodes (x_1 and x_2), one hidden fully connected layer with three nodes (h_1, h_2 and h_3) and an output layer with two nodes (y_1 and y_2).

A FNN, also known as a multilayer perceptron, consists of layers of interconnected neurons, each layer processing information and passing it to the next layer. As the name suggests, the layers form a network, which consists of functions in chain structures. For instance, a network consisting of three functions, or layers, f^1, f^2 and f^3, can look like this [21]:

$$f(x) = f^3(f^2(f^1(x))) \tag{2.7}$$

The depth of the network is defined by the number of functions in the chain. Further, the network has an input layer, x, one or more hidden layers, h, and an output layer, y. FNNs process data in a forward direction, meaning information flows from the input layer through the hidden layers to the output layer. Each neuron in a layer takes inputs, applies weights, w, to those inputs, computes a weighted sum, adds a bias term, b, and then passes the result through an activation function, a. This process is repeated layer by layer until the final output is generated. The general formula for calculating one node in a FNN follows. Here, y_k is an output node for a certain class, k, b is the bias, a is the activation function, while x_i and w_i point to the nodes on the previous layer and the applied weights, respectively.

$$y_k = a(b + \sum_{i=1}^{N} x_i w_i) \tag{2.8}$$

Further, an example is given below, in which the hidden node h_1 from Figure 2.2 is calculated.

$$h_1 = a(b + \sum_{i=1}^{2} x_i w_i) = a(b + x_1 w_1 + x_2 w_2) \tag{2.9}$$

2.2.2 Convolutional Neural Networks

In essence, CNNs are ANNs that use convolution in place of general matrix multiplication in at least one of their layers [21]. As opposed to other ANNs, CNNs are especially well suited for image classification. Generally, a CNN consists of a series of stacked layers of different kinds. Typically, one or more layers consisting of three sub-layers, or stages, are used. In the first stage, a number of convolutions are performed in parallel, producing a set of linear activations, or activation maps, proportional to the input [22][23]. In the second stage, each of these activations are fed to an activation function in order to introduce non-linearity [24]. In the third stage, a pooling operation is performed [21], to reduce the data to lower dimensionality. The next three sections explain each stage in more detail. The output of the last convolutional layer is flattened into a 1D array and fed to one or more fully connected layers, (alternatively with one or more dropout layers inserted in between them) to obtain the final classification result [25]. See Section 2.4 for an explanation of dropout.

2.2.2.1 Convolutional Layer

In a convolutional layer, one or more filters, also called kernels, are slid through all spatial locations of an input image. The convolution operation is displayed in Formula 2.10. Throughout the sliding operation, the dot product between the filter and the image is computed for all individual locations. These resulting dot products produce a feature representation of the original input. Each filter applied to the input image generates an output image, a so-called activation map or filter bank [26].

$$S(i,j) = (I * K)(i,j) = \sum_m \sum_n I(i-m, j-n) K(m,n) \tag{2.10}$$

Above is the convolution formula, where I is a two-dimensional image and K is a two-dimensional filter/kernel. For simplicity reasons, it is more common to use the cross-correlation operation, which is the same as convolution, but without flipping the kernel [21]. The operation changes to the following equation:

$$S(i,j) = (I * K)(i,j) = \sum_m \sum_n I(i+m, j+n) K(m,n) \tag{2.11}$$

The values within the kernels of the convolutional layers are learned by the model during training. Depending on the values, each kernel detects a certain shape in the input image [27]. Additionally, the size of the kernel decides the size of the shapes that are being detected during the convolution, meaning a larger kernel size may be better at detecting larger shapes, while a smaller size may be better at detecting smaller shapes, like textures and finer details, in the input image.

$$n_{out} = \frac{n_{in} + 2p - k}{s} + 1 \tag{2.12}$$

Depending on the individual case, the convolutional operation is given specific parameters by the operator, such as kernel size, image zero padding and stride. Among other things, these parameters determine the resolution of the re-representation, as well as the number of trainable parameters for that specific stage. Formula 2.12 computes the dimensions of the output images, n_{out}, from the convolution operation for the input dimensions, n_{in}, where the stride is s, the padding is p and the kernel size is k.

2.2.3 Activation

An activation function is a function used for calculating the output of a node. It maps the node result into a certain value range. It further adds non-linearity to the network which enables output complexity [28]. Without the non-linear activations, the entire fully connected layers collapse to a simple matrix multiplication. In CNNs, using Rectified Linear Unit (ReLU) as activation function is a popular choice due to its efficiency and strong generalization properties, in addition to its ease of optimization [21].

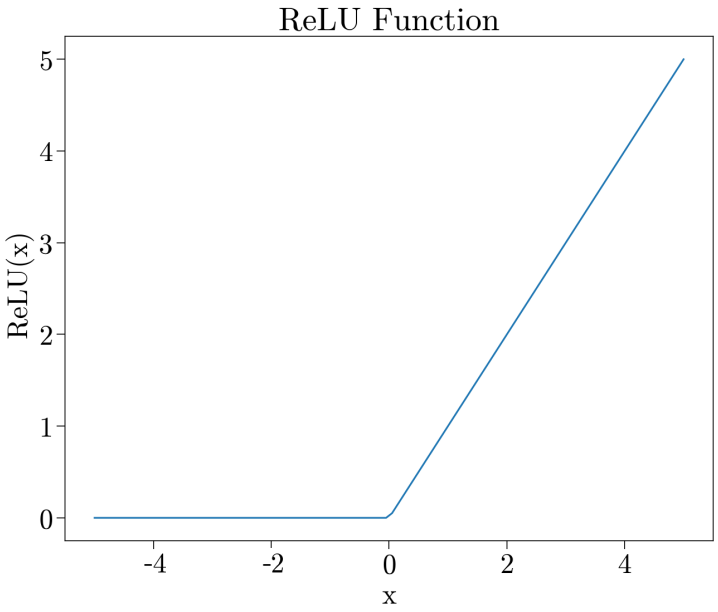

Figure 2.3: ReLU activation function.

Figure 2.3 displays the plot of a ReLU function. In essence, what this activation function does is transform all negative values to 0. By applying ReLU to the convolutional layer, all negative values resulting from the convolution are converted to 0, while the positive values are kept unchanged. The formula for ReLU is presented below, where x is any input value:

$$f(x) = \max\{0, x\} = \begin{cases} x & \text{if } x > 0, \\ 0 & \text{otherwise.} \end{cases} \tag{2.13}$$

2.2.3.1 Pooling Layer

In the pooling stage, or layer, the pooling operation extracts the main features generated by the convolutional layer of a model. This operation both reduces dimensionality and makes the model more robust to positional variations of the input image features [21]. By filtering the less important features and keeping the most representative ones, the model will be able to generalize better, and the risk of overfitting[1] will be reduced. The following figure displays a popular variation of the pooling operation called max pooling [29]. The principle is that by outputting the maximum value of a specified region and discarding the other values, the resulting size of the output feature maps is reduced. Additionally, this reduction causes a decrease in computational needs. Typically, the pooling operation is performed for regions of size 2 x 2 [30]. Similarly to the convolutional layer, Formula 2.12 can be used in the pooling layer for computing the output from the pooling operation.

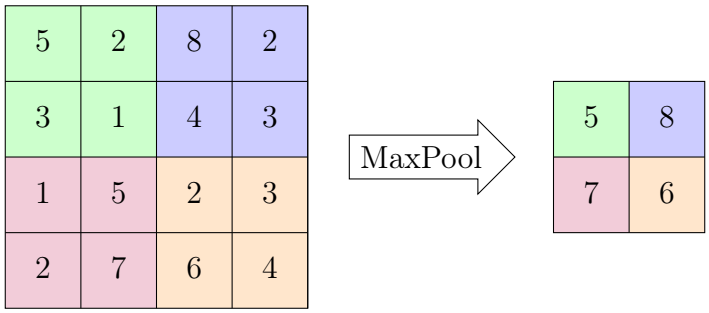

Figure 2.4: Visualization of a MaxPool operation, where the maximum value from each region is retained.

[1] Overfitting occurs when a function fits too well with the training data (ie. it memorizes each data sample instead of the data patterns, causing generalization error).

2.3 Normalization

Data normalization is often performed prior to training of a neural network, in order to ensure stability and to avoid exploding or vanishing gradients, which can slow down or prevent optimal training [31]. There exists various normalization techniques, and a common technique is presented in Formula 2.14, where x is the data point, x_{norm} is the normalized data point, μ is the mean of the dataset and σ is the standard deviation of the dataset. As displayed, the normalization is performed by subtracting the mean of the dataset and dividing each image by the standard deviation of the dataset. This results in data that is centered around 0, with a standard deviation of 1.

$$x_{norm} = \frac{x - \mu}{\sigma} \tag{2.14}$$

2.4 Model Training and Hyperparameter Tuning

A common way to train an ANN is to split the dataset into three subsets; for training, validation and testing. The training set is used to train the model and update its parameters, or more specifically its weights and biases. During training, the process is evaluated with an unbiased validation set. The evaluation metrics can then be examined by the operator for use in the tuning of the model hyperparameters in an attempt of achieving the most optimal model performance. After the tuning process, the model with its final weights and biases can be tested with the last subset which has not yet been seen by the model. The final evaluation of the model with the test set gives an indication of how well the model generalizes, i.e. how well it handles new data [32].

In order to make the model training as efficient as possible, tuning of the different model hyperparameters should be performed. The model hyperparameters are the parameters that determine which weights and biases the model learns [33]. In order to manually tune the model in the most optimal way, it is important to have an understanding of the relationship between hyperparameters and computational resources, as well as training and generalization error [21]. The tuning process is commonly based on the results from model evaluation metrics, which are explained in Section 2.5. In the following, some commonly examined hyperparameters are listed, before they are explained below.

- Train, test and validation set size
- Optimization technique
- Loss function
- Learning rate
- Regularization techniques
- Number of epochs
- Batch size
- Model configuration
  - Activations
  - Kernel parameters
  - Pooling parameters

Train, test and validation set size
Commonly, the training, validation and testing set ratio in a neural network is set to around 70 - 80 % for the training set, 10 - 20 % for the validation set and 10 - 20 % also for the testing set. Depending on the nature of the specific problem, the optimal subset ratio may differ. By the use of evaluation metrics, in particular learning curves, on the training and validation sets of the training phase, it is possible to make some considerations of whether the validation or training set size is unrepresentative [34]. See Section 2.5.1 for theory on learning curves. Based on these considerations, one can then change the ratio accordingly in order to improve the model performance.

Optimization technique
When choosing the ideal optimizer for a specific problem, the decision can be based on the explanation given further down in Section 2.4.2.1.

Loss function
When choosing the ideal loss function for a specific problem, the decision can be based on the explanation given further down in Section 2.4.1.

Learning rate
When choosing the ideal loss function for a specific problem, the decision can be based on the explanation given further down in Section 2.4.2.1.

Regularization techniques
Regularization techniques are commonly employed as a way of improving stability and overfitting prevention. Methods like L2 regularization, early stopping and dropout are some popular choices of regularization technique. L2 regularization (also called weight decay) is a method where a penalty is added to the loss function, based on the size of the weights (larger weights cause larger penalty) [35]. By implementing early stopping on the training phase, the training can stop when a certain criteria is met: often when the loss stops decreasing. Loss is explained in Section 2.4.1. For the Dropout method, some connections between nodes in fully connected linear layers are randomly switched off during training iterations, enhancing the updates of the remaining weights.

Epoch number
Deciding the ideal number of epochs for a specific problem is important in order to achieve the optimal performance and to prevent over- and underfitting. Depending on for instance the learning rate and the complexity of the data, the optimal number of epochs differs. Evaluation using learning curves can help the operator decide the optimal epoch number. See Section Section 2.5.1 for theory on learning curves. For instance one can use the point in the loss curves where the validation loss is starting to deviate from the training loss as a guide for when to stop the training. Further, as mentioned above, using early stopping during training can help to prevent overfitting.

Batch size
There are several considerations to make when deciding the model batch size. The batch size is the amount of samples that the model learns from before it makes an update to the weights and biases. The more batches you have, the less samples per batch, and the more updates the model will make to the weights and biases per epoch, ie. batch size = training dataset / batch number.

The batch (or mini-batch) size can be understood as a trade-off between performance and computational need. Larger batch sizes usually need less time for training, which can be explained by an increase in effectiveness, meaning larger portions of the data are used each time the gradient is updated, leading to fewer overall updates [36]. However, results with larger batch sizes may be less optimal than with smaller batches. One can say that larger batch sizes degrade model generalizability, which has been observed by among others Keskar et al. [37]. A possible explanation of this can be that the many training samples within the same batch interfere with each other's gradient, causing them to cancel each other out [38]. A smaller batch size on the other hand allows the model to learn more from each individual sample, as there are fewer samples in each update. However, a smaller batch size may be more susceptible to random fluctuations in the training data, as opposed to larger batch sizes. Moreover, small batch sizes can offer a regularizing effect [21]. For reference see explanation of regularization above. Finally, the batch size may also be limited by the computing power of the given GPU.

Model configuration
The model configuration is an important hyperparameter. Three important factors are the model depth (number of hidden layers), width (number of nodes/filters in hidden layers) and complexity in relation to data amount/complexity. As the complexity of the data increases, the need for a deeper network is required [39]. It follows then that the requirements of the dataset size is increased with increased network depth. Earlier explanations cover considerations regarding model activations, as well as kernel and pooling parameters, see Section 2.2.2.

2.4.1 Loss Function

The loss, or equivalently the cost, of the model is defined as the difference between ground truth, which are the labels of the input, and the predicted output. For classification tasks, the cross-entropy loss function is commonly used. If i is the class index ranging from 1 to N, y is the true label, while $\hat{y}$ is the predicted label, then the cross-entropy loss function L can be expressed like this:

$$L(y, \hat{y}) = \sum_{i=1}^{N} y_i \log \hat{y}_i \tag{2.15}$$

Further, the average cross-entropy loss can be defined for a batch, B, where j is the index for samples within the batch:

$$\overline{L}(y, \hat{y}) = \frac{1}{B} \sum_{j=1}^{B} L(y_j, \hat{y}_j) \tag{2.16}$$

2.4.2 Backpropagation Algorithm

The learning of neural networks requires computation of the gradients of complex high-dimensional loss functions and updating of the network parameters in direction of the negative gradient direction. This can be achieved by applying the back-propagation algorithm [21]. According to Rumelhart et al., backpropagation repeatedly adjusts the weights of the connections in the network so as to minimize the difference between the actual and the desired output vector of the network [40]. By using the chain rule of calculus displayed in Formula 2.17, the backpropagation algorithm can propagate the prediction error backwards into the network, layer by layer, to single weights. The chain rule of calculus states the following [21]:

$$\frac{dz}{dx} = \frac{dz}{dy}\frac{dy}{dx} \tag{2.17}$$

In the above equation, x is a real number, and the functions $y = g(x)$ and $z = f(g(x)) = f(y)$ are both mapping from and to real numbers.

2.4.2.1 Optimization and Gradient Descent

A common technique for updating the weights and biases of a network is gradient descent. Here, the updated weights are computed iterativeley, while minimizing the loss until convergence is reached. One step of the process can be defined as:

$$w' = w - \epsilon \nabla_w L(w), \tag{2.18}$$

where w is the initial weight, w' is the updated weight, ϵ is the learning rate, and $\nabla_w L(w)$ is the gradient of the loss function with respect to the model parameters. The learning rate determines the size of the steps taken during each iteration, deciding the movement speed down the slope. When the gradient vanishes, a local or global minimum is found [21].

As displayed in Formula 2.18, the choice of learning rate is an important factor in regards to the performance of gradient descent. If the learning rate is too small (and there are too few epochs), the algorithm may not reach convergence. However, if the learning rate is too large, it can cause the training to overshoot and miss the optimal convergence point, resulting in an increased loss.

There are different variants of gradient descent commonly employed in deep learning networks. These variants are often called optimizers, and Stochastic Gradient Descent (SGD) is one of the most popular of these. Here, the updates are computed using a

random subset of the training data, or more specifically using mini-batches, in each iteration. This saves time, as the model only needs to compute the gradient for each mini-batch of input data, rather than for every individual example. Moreover, computing the gradients on mini-batches yields more stable and less noisy gradients, compared to the gradients of single data samples. Additional popular optimizers include Adam, Adagrad and RMSprop, which adapt the learning rate during training [21]. This adaptation helps the algorithm to not overshoot the local minimum, by decreasing the learning rate based on the steepness of the gradient.

2.5 Model Performance Evaluation

Model training can be evaluated in a number of ways. In order to evaluate the performance of a model on given data, it is common to measure the accuracy or equivalently, the error rate of the system [21]. These scores are commonly evaluated by plotting and interpreting learning curves. Other popular methods for multi-class problems include using confusion matrices, which are computed on the test set. From these matrices, precision/recall metrics can be computed.

2.5.1 Learning Curves

Generally speaking, learning curves display predictive performance relative to the amount of learning effort [41]. It is common to plot curves for the training vs. validation loss and accuracy per training epoch. The relationship between the curves can give valuable information about the training. For instance, the curves can be used as an assessment tool for overfitting, where the relationship between the training vs. validation curves for both loss and accuracy gives an indication of whether the model overfits (large gap between curves) or not (no gap) [42].

2.5.2 Confusion Matrices and Precision/Recall Scores

Confusion matrices are commonly computed from the testing set (data unseen by model during training).

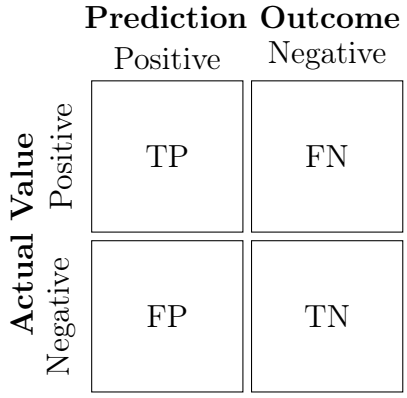

Figure 2.5: T = True, P = Positive, F = False and N = Negative. The prediction outcome indicates the predicted values, while the actual value indicates the true labels.

A True Positive (TP) score indicates correct predictions of the positive truths. True Negative (TN) indicates correct predictions of the negatives. On the other hand, False Positive (FP) indicates incorrectly predicted truths (ie. model predicted a negative truth to be positive). False Negative (FN) indicates incorrectly predicted negatives (ie. model predicted a positive truth to be negative).

Out of all the positive predicted, the precision score indicates what percentage is truly positive. Out of the total positive, the recall score indicates what percentage are predicted positive.

$$Precision = \frac{TP}{TP + FP}, \qquad Recall = \frac{TP}{TP + FN} \tag{2.19}$$

The accuracy is the number of correct predictions divided by the total number of predictions.

$$Accuracy = \frac{TP + TN}{TP + TN + FP + FN} = \frac{TP + TN}{Total} \tag{2.20}$$

CHAPTER 3

METHODS

In this chapter, the methods applied in this research are described and explained in-depth. These methods were integrated into the workflow leading to the research results presented in Chapter 4, and include the full process from the data collection, to the CNN evaluation and live-classification. The flow chart in Figure 3.1 below displays the general workflow of the research, with its core steps laying the foundation for this chapter.

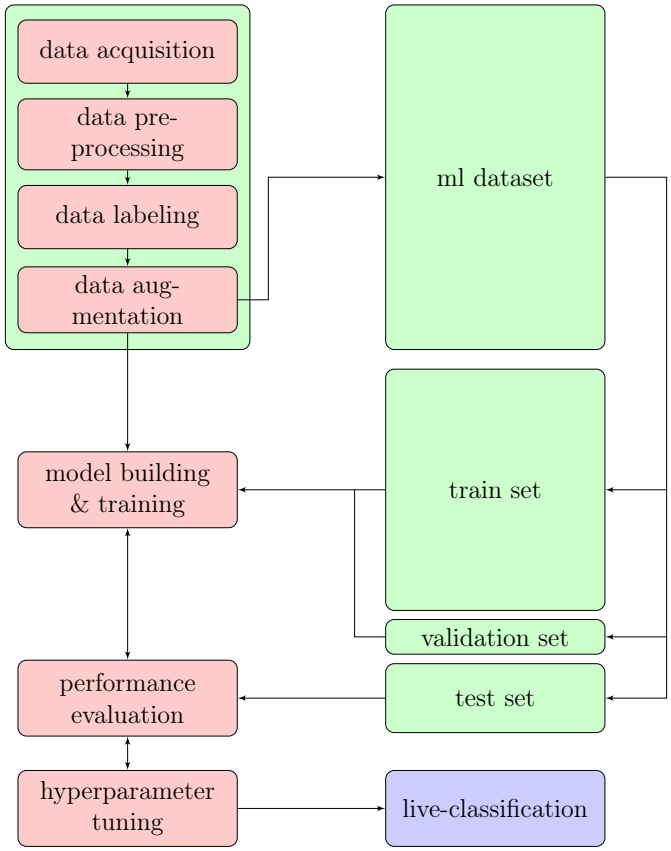

Figure 3.1: Project workflow visualization. The blocks colored in red are the core steps of the project. The green blocks represent the data blocks, while the blue block represents the final model deployment phase. Notice the bidirectional arrows in the bottom red blocks.

3.1 Data Acquisition

The data in this research was acquired using the OptoDAS IU from Alcatel Submarine Networks[1]. The IU was placed at Marienborg station in Trondheim, interrogating a fiber co-located with the 51 km long railway section between Trondheim and Støren. See Figure 3.2 for a map of the fiber placement. The entire dataset of the experiment contains continuous data collected through one week dated back to August 2021. This research explores 70 minutes of DAS data during one day of that week, collected at a sampling frequency of 2 kHz. The fiber length is 51 km, which implies that there are a total of 51 km / 1.02 m = 50,000 absolute channels along the fiber segment. In order to improve the SNR the channels were stacked by a factor of four during the acquisition, yielding 12,500 channels with a spatial sampling interval of dx_{acq} = 4 SSI = 4.08 m [43]. To ease the data handling without losing too much information the channels were further stacked by a factor of eight, resulting in a final spatial sampling interval of dx = 32.6 m and 1,563 channels. The gauge length (see Formula 2.4) was set to twice the initial spacing; 2 * 4.08 m = 8.16 m.

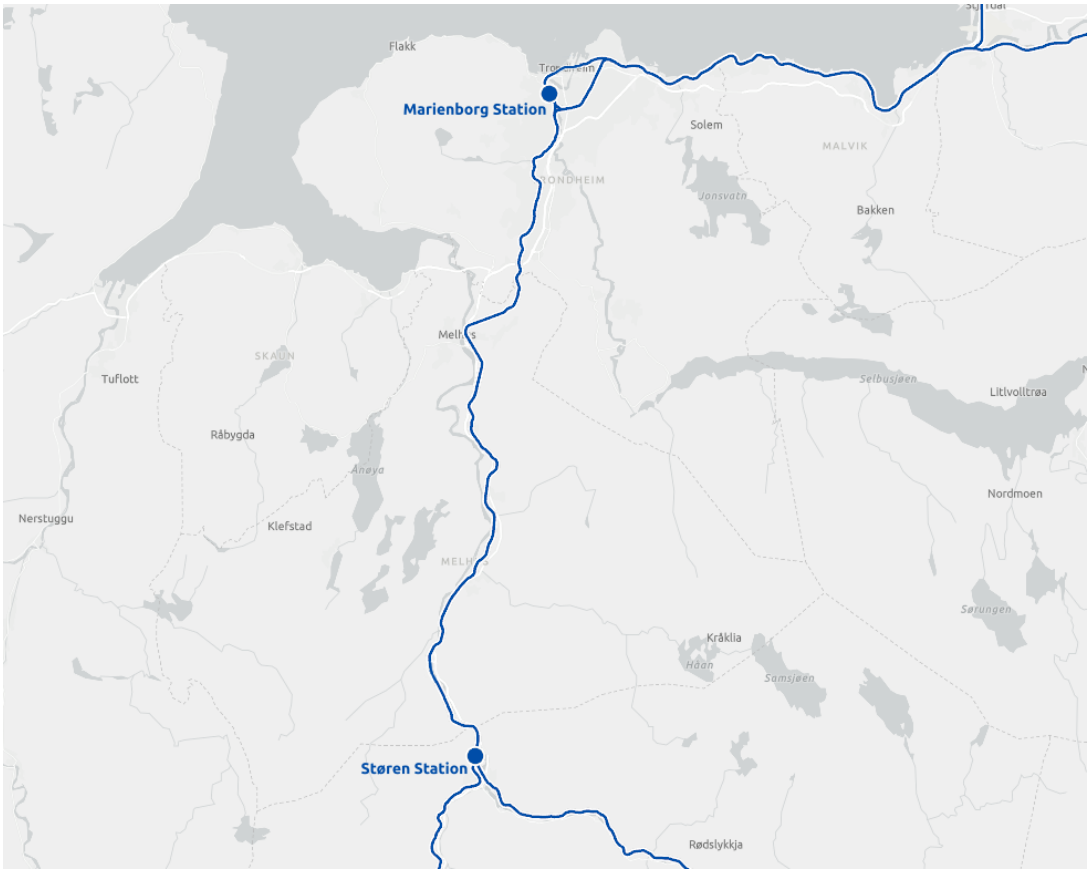

Figure 3.2: Map of railroad, courtesy of `https://togkart.banenor.no/`. Data collected between Marienborg and Støren.

[1] `https://web.asn.com/en/`

3.2 Data Preprocessing

The raw data was first converted from time-differentiated phase to strain rate and integrated along the time axis to obtain strain, see Formula 2.6 for reference. Thereafter, the data was downsampled to 500 Hz, the linear trend was removed from the data and a fifth order bandpass filter was applied. Different frequency ranges were tested as illustrated in Figure 3.3. The higher frequency band of 25 - 150 Hz was chosen since a lot of the low frequency noise around the main signal was removed and it appeared clearer for the classification. With a high cut frequency of 150 Hz and a sampling frequency of 500 Hz it is ensured that the Nyquist theorem requirement is met [44].

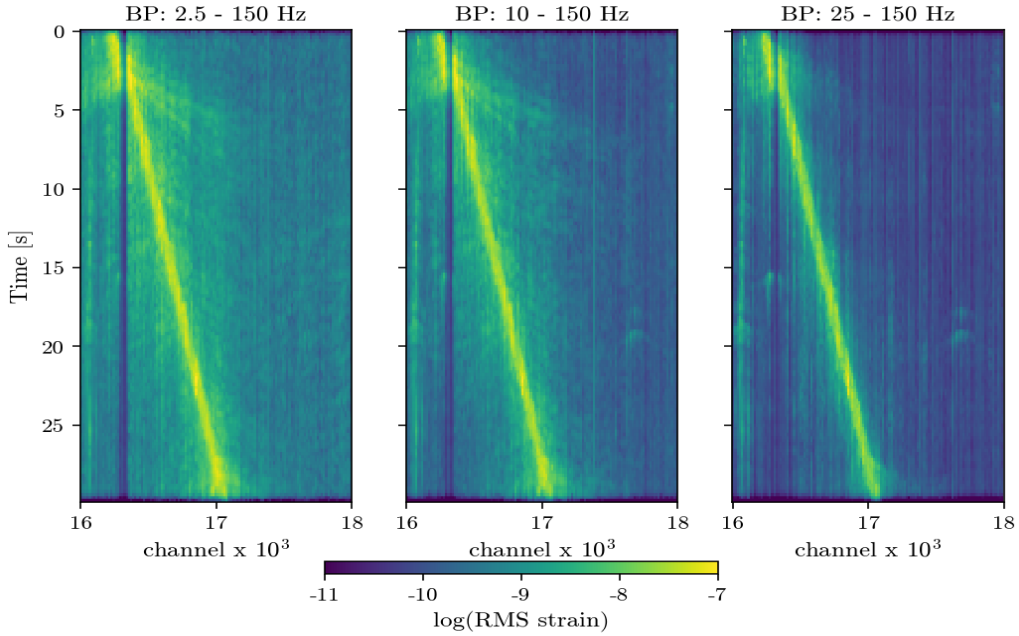

Figure 3.3: Comparison of different bandpass filters applied to the DAS data.

Instead of using the raw strain data with both positive and negative amplitudes (compression and extension of the fiber), its rolling Root Mean Square (RMS) was computed, yielding sufficient information of the average absolute amplitude while drastically reducing the amount of data to handle. The rolling RMS is computed as:

$$RMS_j = \sqrt{\frac{1}{N_w} \sum_{i=1}^{N_w} d_i^2}, \tag{3.1}$$

where d_i is the strain amplitude at index i and N_w is the number of samples within the window. The RMS was computed on each channel separately with a window length of 0.4 s with 50 % overlap, resulting in one RMS sample at a new time index j every 0.2 s. Finally, the dataset (70 mins x 51 km) was cut into 8,160 windows of 1 min and 1.5 km extent with 50 % overlap in both time and space direction. This approach allows for later live-classification of events every 30 s at a spatial interval of 750 m along the entire line. Moreover, prior to the data labeling, the windows were normalized, see Section 2.3 for reference.

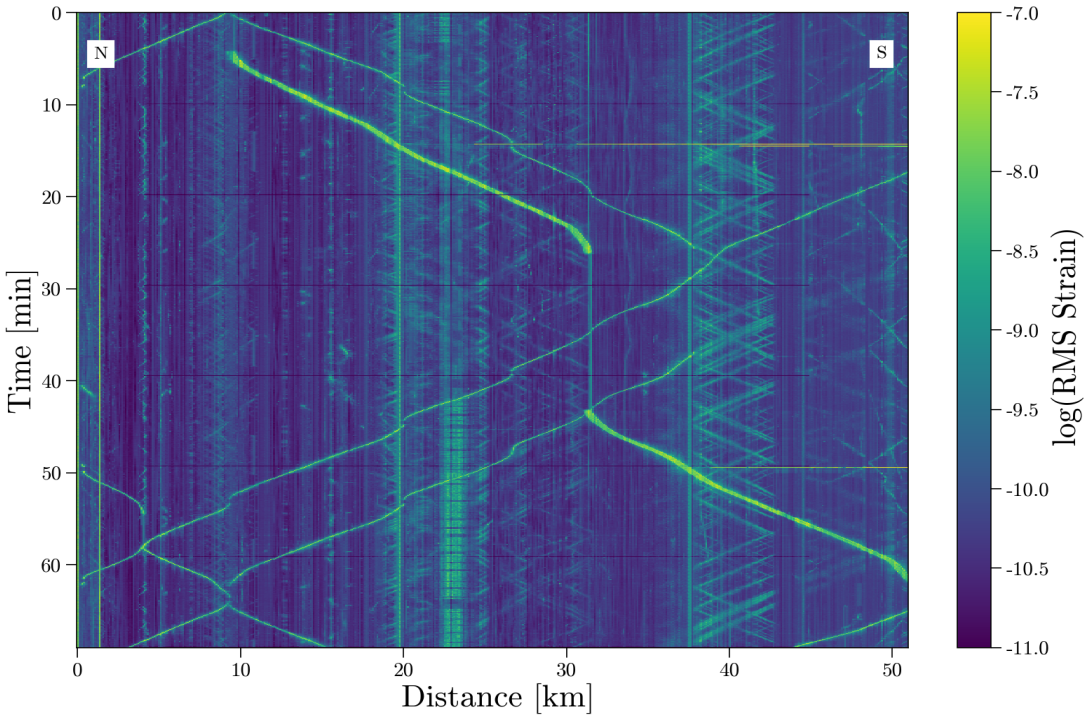

Figure 3.4: RMS strain dataset alongside the 51 km railway line section. N and S mark North (Marienborg, Trondheim) and South (Støren), respectively.

3.3 Data Labeling

The data labeling process was performed by manual inspection of the signals. Following, a list of classes chosen for the dataset is presented. The characteristics of each class are explained. Further, considerations made prior to the labeling process are discussed. For reference, see Figure 3.4 and 3.5 for the full 70 minutes of data along the 51 km length railway section, before and after the labeling process. See Table 3.2 for a view of the number of labeled windows per class. Each of the 12 signal classes was selected based on which events in the dataset were identified, as well as their potential usefulness for the end user.

Table 3.1: Table displaying index and its corresponding class. Notice the directions marked North to South (NS) and South to North (SN).

0	Ambient noise	**6**	Cargo train NS
1	Unknown noise	**7**	Cargo train SN
2	Artifact noise	**8**	Motor vehicle NS
3	Stationary noise	**9**	Motor vehicle SN
4	Passenger train NS	**10**	Motor vehicle NS, SN
5	Passenger train SN	**11**	Multiple objects

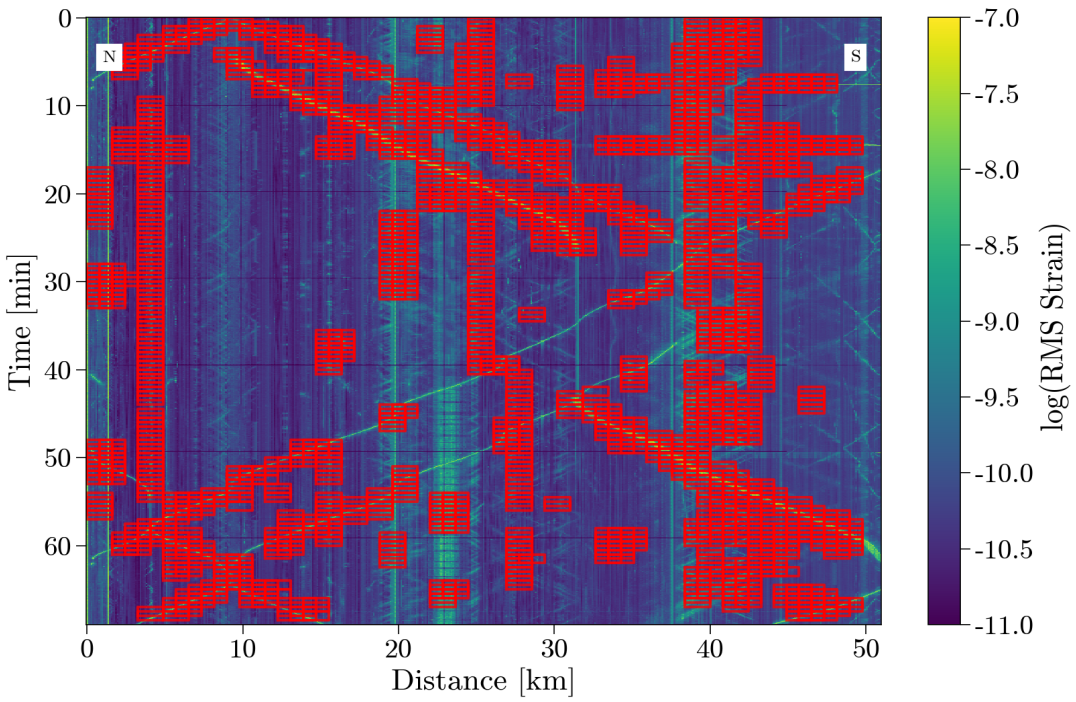

Figure 3.5: RMS dataset as in 3.4 with labeled data windows marked in red.

The **passenger train** classes include signals coming from passenger trains. Passenger train signals are characterized by thin and long diagonal high amplitude lines. In Figure 3.4, multiple of these can be seen. One example is the line beginning at 0 km and approximately 62 minutes. Following this line one can see it ending at 51 km and approximately 22 minutes. The curves in these signals are caused by velocity changes and stops at train stations along the way. See Figure 3.6 for a detailed view of what a passenger train signal can look like.

Table 3.2: Amount of labeled images per class. Total amount of labeled images is 1667.

Idx.	**Class**	**Labeled**	**Idx.**	**Class**	**Labeled**
0	Ambient noise	188	6	Cargo train NS	159
1	Unknown noise	221	7	Cargo train SN	0
2	Artifact noise	64	8	Motor vehicle NS	114
3	Stationary noise	147	9	Motor vehicle SN	138
4	Passenger train NS	123	10	Motor vehicle NS, SN	175
5	Passenger train SN	163	11	Multiple objects	149

A similar approach was followed for the **cargo train** class. The class is characterized by a similar diagonal line as the passenger train signal, however this line is wider, i.e. the signal affects multiple channels simultaneously, and has a slightly higher amplitude. Again looking at Figure 3.4, a cargo train signal is visible as the somewhat wide, high amplitude line, ranging from approximately 10 km and 5 min to 51 km and 65 min. The "jump" at 30 km is caused by the cargo train stopping. See Figure 3.6 for a detailed view of what a cargo train signal can look like. Notice that there are no cargo trains going from South to North in the dataset.

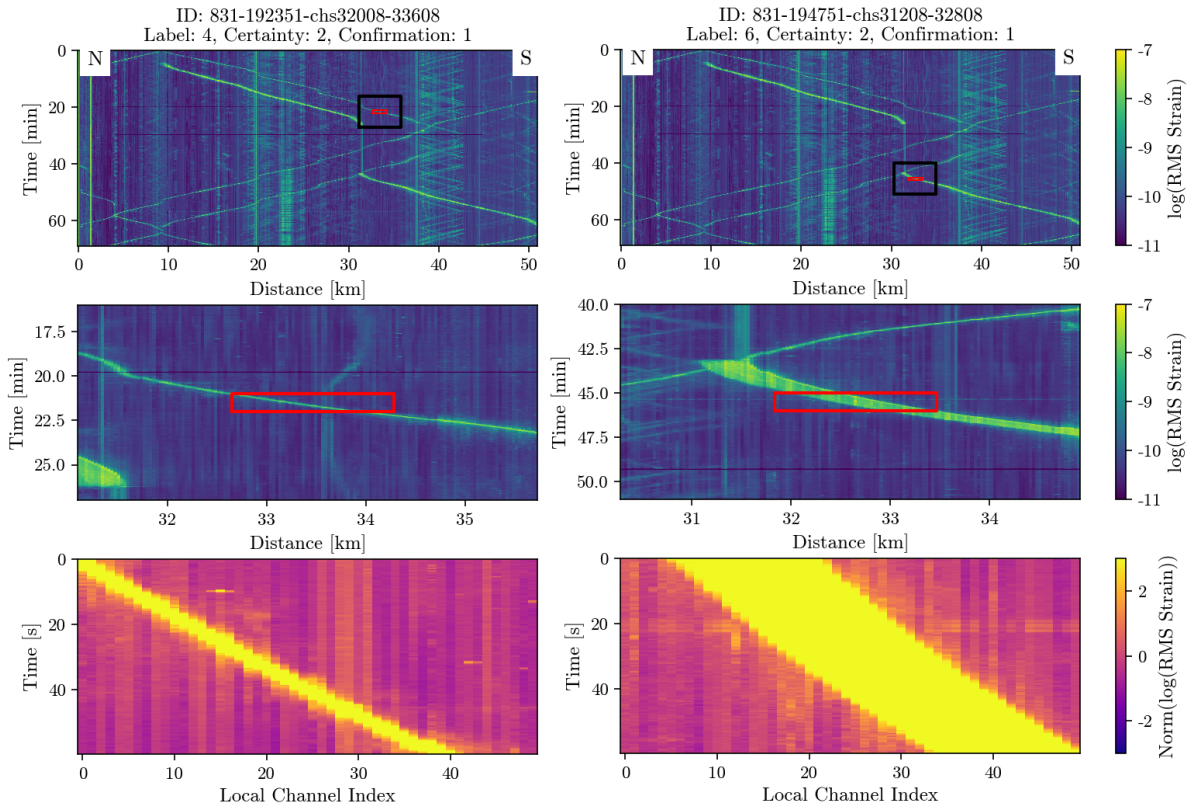

Figure 3.6: To the left: passenger train NS. To the right: Cargo train NS.

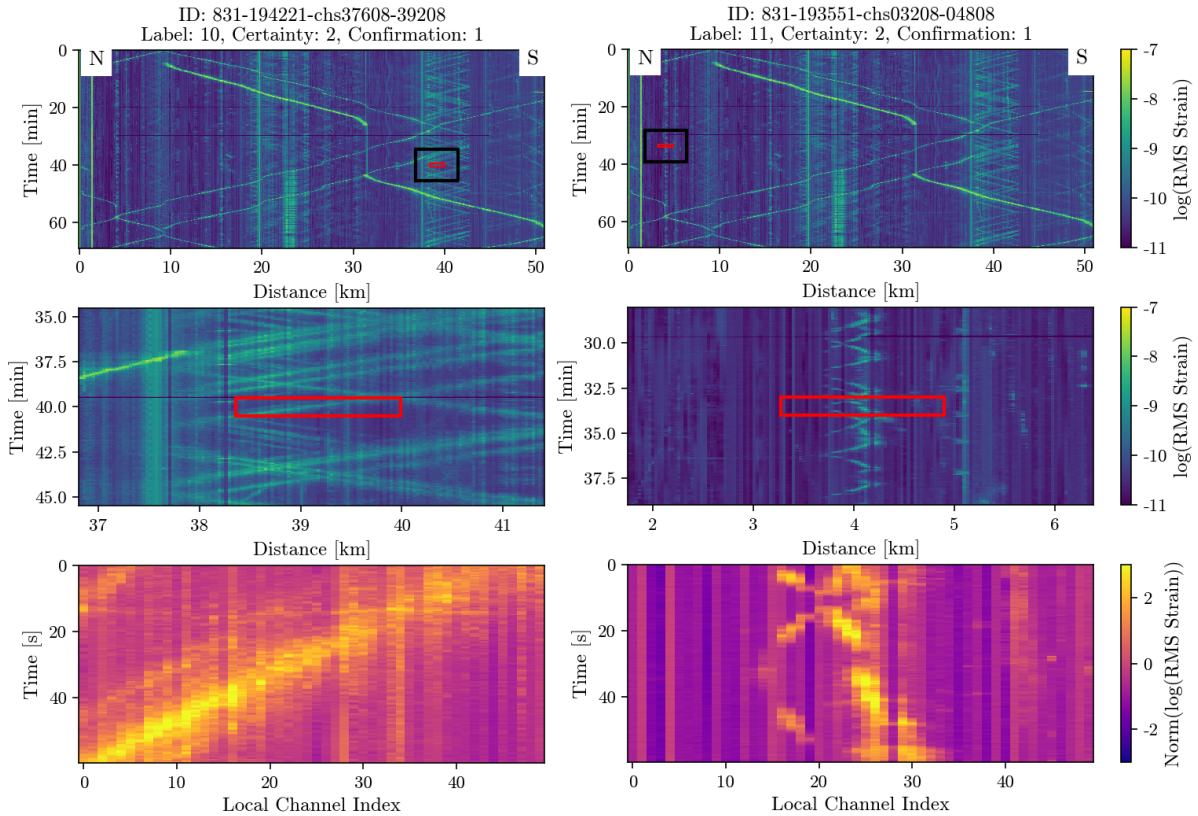

Figure 3.7: To the left: Vehicle SN. To the right: Vehicle crossing.

Further, all signals coming from motorized vehicles are included in the **motor vehicle** classes. These signal classes are, similarly to the train signal classes, characterized by diagonal lines. However, these usually have a lower amplitude and are apparent only in certain channel ranges (around crossings or roads close to the train track). A subclass within the motorized vehicle class consists of the crossings between the railway line and vehicle road. As opposed to the vehicles traveling on roads parallel to the railway line, these are cars traveling east-west and west-east. These signals are characterized by short diagonal lines in both directions. The classes are divided into northbound and southbound directions, as well as both northbound and southbound directions in the same frame. See Figure 3.7 for a detailed view of what two motor vehicle signals can look like.

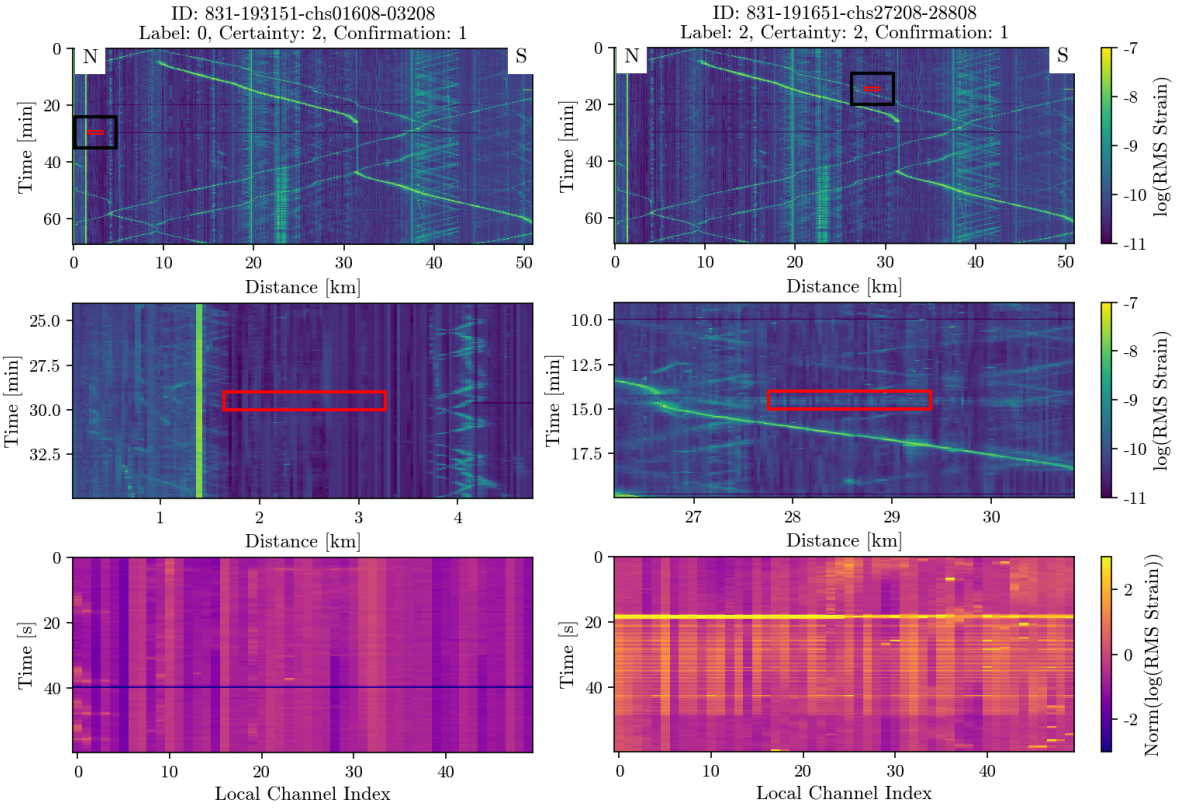

Figure 3.8: To the left: Ambient noise. To the right: Artifact noise.

The data further features several noise classes. The **ambient noise** class consists of more or less constant background noise, where little to nothing happens in each frame. See Figure 3.8 for a detailed view of what an ambient noise signal can look like. **Artifact noise** represents noise signals coming from the IU, which is characterized by thin, horizontal, high amplitude lines on certain times, ranging over sections of channels across the dataset. See Figure 3.8 for a detailed view of what an artifact noise signal can look like. There is also **stationary noise**, which is characterized by vertical lines in the dataset, differing in amplitude and channel width, usually ranging through the entire time axis. See Figure 3.10 for a detailed view of what a stationary noise signal can look like. The **unknown noise** class includes the rest of the signals that are unknown to the operator. This can include construction noise (if not constant), or other unknown events. See Figure 3.9 for a detailed view of what two different unknown noise signals can look like.

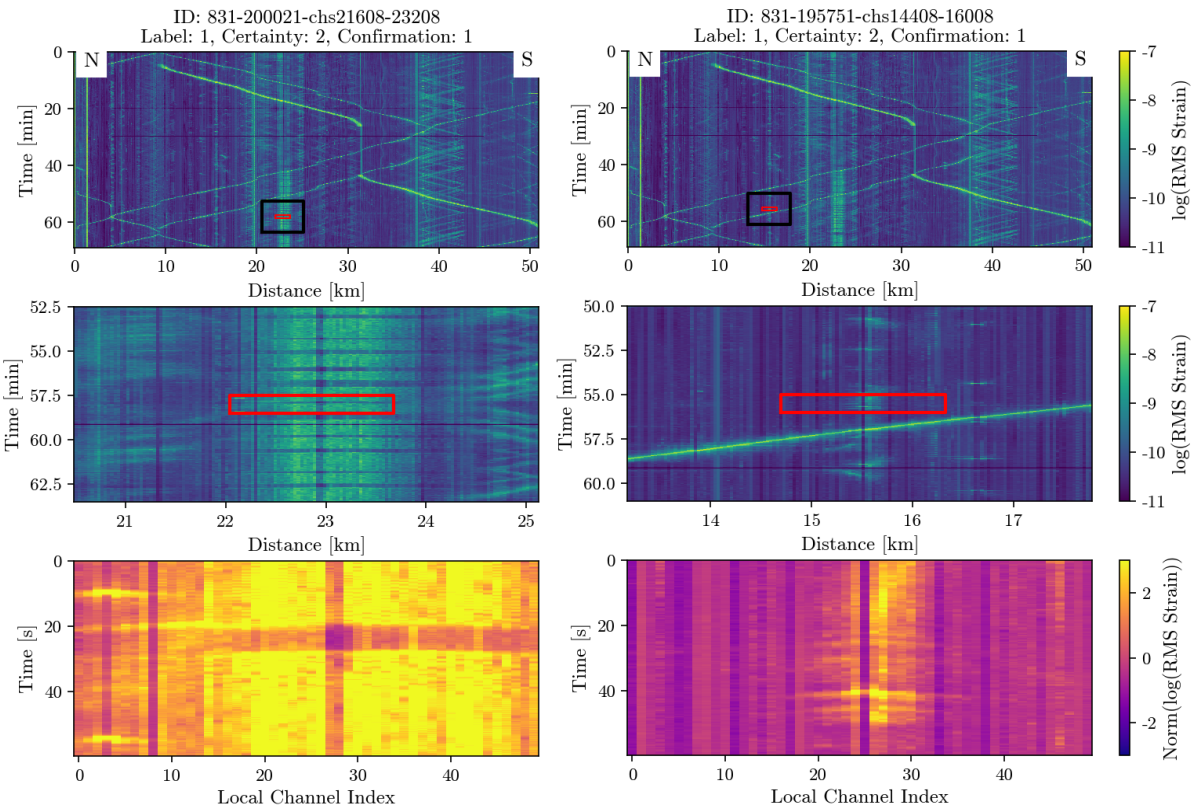

Figure 3.9: Two examples of unknown noise.

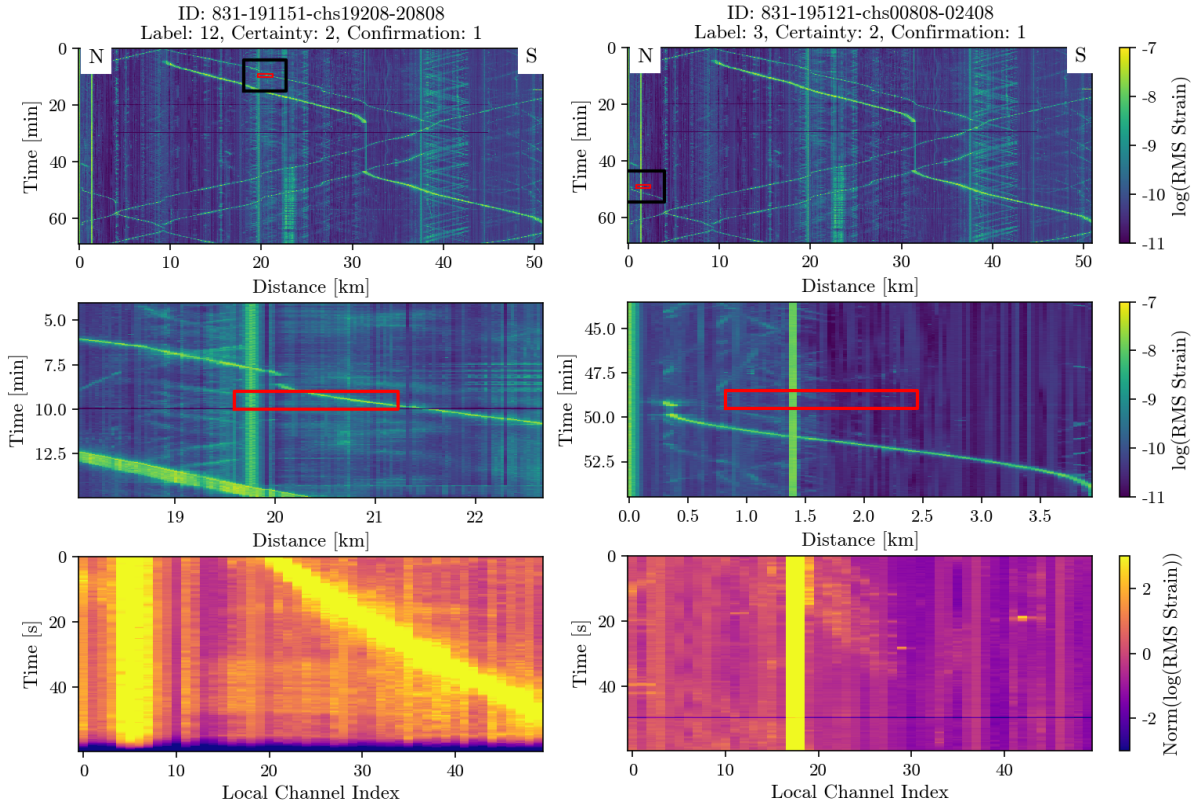

Figure 3.10: To the left: Multiple objects (stationary noise and cargo train NS). To the right: Stationary noise.

Lastly, the frames that contain more than one of the above signals (with similar amplitude), were labeled as **multiple objects**. This class also includes train stops. See Figure 3.10 for a detailed view of what a multiple objects signal can look like.

In order to make the labeling process more efficient, a labeling script was created to enable semi-automatic inspection and labeling of each image. Using the labeling script, the entire dataset can be plotted in one image. Based on this image, channel and time indexes were selected in such a way that the user could choose a class label based on what was displayed in that specific time and channel index. Additionally, for each label, the script prompts for a certainty, ranging from 0 (uncertain) to 1 (somewhat certain) and to 2 (certain). Lastly, the labels were either confirmed or changed by a second operator in order to reduce the human bias and the errors in the dataset.

3.3.1 Labeling Considerations

During the labeling process, the label for each window was chosen based on which signal class displayed the highest amplitude within said window. Thus, windows containing multiple objects were labeled as multiple objects unless one of the signal types was notably more apparent than the other(s).

Additionally, there was a focus on acquiring an even and balanced dataset. An approximately equal number of windows were labeled per class (except "cargo train SN"). Besides, there was also a focus on labeling as many individual events as possible, given all events occurring in the dataset.

Moreover, the signals were only labeled when clearly visible in the image. Accordingly, if the identifiable part of the signal class was obscured (ie. signal was on an edge or in a corner) it was ignored, and the remaining signal within the image was labeled instead. As the real-time classification would be performed over longer time instances than the time instances of the labeled images, the ignored signal would still be detected and classified in an adjacent frame.

Lastly, after the labeling process, a filter with a set amplitude limit was applied to a smoothed version of the labeled images. Images that did not exceed the amplitude threshold after smoothing were subsequently assigned to the ambient noise class.

3.4 Data Augmentation

To balance the distribution among the classes and increase the general size of the dataset, which is important to reduce generalization error of a model [21], each class label was considered for data augmentation. It is good practice to have an even amount of data samples in each class. A rule of thumb states that you would need approximately 5,000 images per class in order to sufficiently train a CNN. To generate this amount of data samples, several data augmentation techniques can be used. According to Goodfellow et al., classifiers can benefit from random translations, rotations and flips of the data [21].

A consideration made when selecting augmentation techniques for this data, was the attempt to acquire "naturally occurring" results, which might have a chance of happening in real-time. Thus, augmentation techniques such as shifting, flipping and Gaussian blurring and adding noise were chosen.

Prior to augmentation, labels in the original dataset were flagged with a score from 0 to 2 (no, maybe, yes), based on how compatible the window would be with augmentation. For instance, a window displaying a train signal should be as much in the center of the window as possible for it to be compatible with shifting in both directions, while still maintaining its original class label. As a safety measure, after augmentation, these shifts were examined in order to see whether they were still in the correct class, or if the class needed to be changed

The following augmentation methods were applied:

- Inverting (around x axis and y axis)
- Shifting (in time and distance)
- Adding Gaussian noise
- Gaussian blurring

In the first step of the data augmentation, all windows flagged with an augmentation score of 1 (maybe) or 2 (yes) were shifted along both the time and the space axis, in both positive and negative direction. Based on the nature of the data (ie. the dimensions of each image), shifts of ± 7 and ± 15 s in time and ± 7 and ± 15 channels in space direction were used. In this way, each original window would be able to generate up to eight subsequent windows. These nine windows (8 shifted + 1 orig.) formed the base of the windows which in the next step were used for further augmentation. Axis inversions, Gaussian blurring and Gaussian noise were applied. Subsequently, each originally labeled window could generate up to 9 x 8 = 72 data examples. See Table 3.3 for resulting windows in each class after shifting.

After augmentation, the total number of windows in each class was at least 5,000 (see Table 3.3). Then, 5,000 windows were randomly selected from each class, which were used for the classification task. This resulted in a total dataset size of 5,000 * 12 = 60,000 windows.

Table 3.3: Amount of augmented windows after shifting, as well as the amount after final augmentation.

Idx.	**Class**	**Shifted**	**Final**	**Idx.**	**Class**	**Shifted**	**Final**
0	Ambient noise	1137	13555	6	Cargo train NS	1103	6612
1	Unknown noise	1887	13658	7	Cargo train SN	0	6202
2	Artifact noise	577	6221	8	Motor vehicle NS	612	5328
3	Stationary noise	1148	7611	9	Motor vehicle SN	898	6353
4	Passenger train NS	893	7994	10	Motor vehicle NS, SN	1039	7933
5	Passenger train SN	1328	9737	11	Multiple objects	960	8316

3.5 Model Building and Training

Two of the main components of neural networks are the model and the training of the model. The following sections explain the process of developing and training a number of models for the acquired dataset.

3.5.1 Base Model Configurations

The model building was carried out in PyTorch[1], which is a Python[2] coding library for deep learning. The base of all model configurations includes four convolutional layers consisting of the three stages explained in Section 2.2.2. Additionally, the activation function for all convolutional layers was selected to be ReLU. Moreover, all model architectures feature a flattening and fully connected layer in the end. Besides, for each architecture the very first three-stage layer had the pooling layer removed. This was done in order for the dimension of the original input to decrease enough before it eventually was passed into the first pooling stage. As the pooling operation commonly divides the dimensions by two, the goal was to adjust the dimensions enough so that they would be multiple times divisible by two, or in other words would be applicable for several pooling layers. See Table 3.4 for a detailed view of all model architectures in addition to the input dimensions to each layer. By adjusting the stride and padding parameters for each layer according to Formula 2.12, the image dimensions were decreased from the original 300 x 50 to 37 x 6. For reference, all pooling parameters were equal for all models in this research (padding = 0, stride = 1 and kernel size = 2).

Additionally, two of the base model configurations feature an additional convolutional layer, and the reason for this is explained in Section 3.6. See Table 3.4 for placement reference.

Table 3.4: Architecture of all tested models. The "Increasing" and "Large" models have an extra layer colored in gray. See Section 3.6 for explanation of "Increasing" and "Large". Input denotes the image/feature dimensions.

Layer	**Input**
Conv2d	300 x 50
Conv2d	298 x 48
Maxpool2d	296 x 48
Conv2d	148 x 24
Conv2d	148 x 24
Maxpool2d	148 x 24
Conv2d	74 x 12
Maxpool2d	74 x 12
Flatten	37 x 6
Linear	
Output	12 x 1

[1] https://pytorch.org/
[2] https://www.python.org/

3.5.2 Training

The model training was performed by running a training loop and a validation loop within the same for loop. All training was run on a computer with an AMD Ryzen 9 5900X CPU at 4.26 GHz, and an NVIDIA GeForce GTX 3080 Ti GPU.

3.6 Performance Evaluation and Hyperparameter Tuning

In order to achieve the most optimal model performance, hyperparameter tuning can be utilized. This section will explain the hyperparameter tuning and model evaluation process for this research.

3.6.1 Hyperparameter Tuning

A motive for the hyperparameter tuning process was to cross-test as many hyperparameters as possible. In order to achieve this, a four step tuning process was developed, where each step would optimally lead to better model performance. Following each step of the process, the results were evaluated using maximum validation accuracy and learning curves. See Section 2.5 for theory on evaluation. According to the evaluation, the best results were selected for further tuning in the next process step. Following, a list containing each of the four steps in the hyperparameter tuning process is displayed, before each step is further explained below.

1. **Iterate through models**
2. **Iterate through learning rates and optimizers**
3. **Investigate additional hyperparameters**
4. **Final run and live-classification**

1. Model Iteration

As a first step in the hyperparameter tuning process, several models were trained on a chosen set of initial hyperparameters. The initial hyperparameters were somewhat arbitrarily chosen, but within standard ranges. See Table 3.7 for a display of the hyperparameter values.

- ☐ Train, test and validation set size
- ☐ Optimization technique
- ☑ Loss function
- ☐ Learning rate
- ☐ Regularization techniques
- ☐ Number of epochs
- ☐ Batch size
- ☐ **Model configuration**
  - ☐ **Activations**
  - ☐ **Kernel parameters**
  - ☐ **Pooling parameters**

A total of 10 model architectures were built and utilized for this step. Based on the layer configurations displayed in Table 3.4 in the last section, different combinations of convolutional filter sizes and numbers were tested. In order to achieve a broad selection of model architectures, a foundation of three model types was created; equal filter sizes for all layers, increasing filter sizes and decreasing filter sizes. In the model type consisting of equal filter sizes, a small, medium and large basic model was made up of filter sizes of three, five and seven, respectively. See Table 3.5 for reference. In order to decrease the number of trainable parameters and thereby decrease training time, a convolutional layer with a filter size of 1 x 1 was added in the basic large and the increasing models [21][45].

Table 3.5: Kernel sizes for the different model architectures in this step. For all models, there are two variations in the number of filters; fewer or more. The number of trainable parameters for each model is in the bottom row, where "Fewer" and "More" refer to the amount of filters applied in each convolutional layer, resulting in the given numbers of trainable parameters. See Table 3.6 for number of filters per convolutional layer in each model.

	Basic Small	**Basic Medium**	**Basic Large**	**Increasing**	**Decreasing**
	3 x 3	5 x 5	7 x 7	3 x 3	7 x 7
Filter	3 x 3	5 x 5	7 x 7	3 x 3	5 x 5
Size	3 x 3	5 x 5	7 x 7	5 x 5	3 x 3
/Layer			1 x 1	1 x 1	
	3 x 3	5 x 5	7 x 7	7 x 7	3 x 3
Fewer	1,069,836	1,758,476	778,604	1,444,012	1,103,884
More	2,913,804	5,667,340	2,289,772	5,091,660	3,047,436

For all five models, each model contains two versions, one with fewer filters, and one with more filters. By iterating through these different models, the results were stored and reviewed, before two of them were selected for Step 2 of the tuning process. As explained earlier, the selection of the first model was based on the maximum validation accuracy acquired out of all epochs. However, the selection of the second model was based on the closest relationship between the training and validation accuracy. The assumption was that the second model would not yet have reached training accuracy convergence, thus having the most potential for further learning.

Table 3.6: Filter numbers per convolutional layer in the different models.

	Basic Small		**Basic Medium**		**Basic Large**		**Increasing**		**Decreasing**	
	Fewer	More	Fewer	More	Fewer	More	Fewer	More	Fewer	More
	32	64	32	64	32	64	64	128	32	64
Filter	64	128	64	128	64	128	128	256	64	128
Amount	128	256	128	256	128	256	128	256	128	256
/Layer					32	32	32	64		
	256	512	256	512	64	64	128	256	256	512

Table 3.7 displays the additional hyperparameter values chosen for this step. The model training and tuning was based on a training and validation set, while the final evaluation was based on a test set. The training and test set was first split with a ratio of 9:1, before the training set was re-split into training and validation with a ratio of 8.5:1.5. This resulted in the dataset sizes in the table below. The batch size of 16 for this step was chosen based on computing efficiency and GPU capacity, as a larger number of batches (ie. a smaller batch size) increase training speed. The epoch number in this step was set to be greater than the epoch of validation convergence, in order to capture the maximum accuracy of each model run.

Table 3.7: Additional hyperparameters applied.

Hyperparameter	**Value**
Train set size	76.5 %
Validation set size	13.5 %
Test set size	10.0 %
Optimizer	SGD
Learning rate	0.025
Batch size	16
Regularization	None
Number of epochs	15

2. Learning Rates and Optimizer Iteration

- ☐ Train, test and validation set size
- ☐ **Optimization technique**
- ☑ Loss function
- ☐ **Learning rate**
- ☐ Regularization techniques
- ☐ Number of epochs
- ☐ Batch size
- ☑ Model configuration
  - ☑ Activations
  - ☑ Kernel parameters
  - ☑ Pooling parameters

Step 2 of this process aimed to test different learning rates and optimizers. The two selected models from Step 1 were retested using a nested loop of learning rates and optimizers, with a goal of cross-testing each learning rate with each optimizer. The tested learning rates and optimizers are given in Table 3.8. Table 3.9 lists additional hyperparameter values applied in this step. In an attempt of performance increase, the batch size was increased from 16 to 32 for both models, see Section 2.4 for theory.

The selection of the hyperparameter setup for further tuning in Step 3 followed a similar process as the first model in Step 1. It was selected based on the maximum validation accuracy acquired through all epochs for each training within the nested loop of learning rates and optimizers.

Table 3.8: Learning rates and optimizers.

Learning rate	**Optimizer**
1.0 x 10^{-4}	SGD
5.0 x 10^{-4}	RMSprop
1.0 x 10^{-3}	Adagrad
5.0 x 10^{-3}	Adam
1.0 x 10^{-2}	
2.5 x 10^{-2}	
5.0 x 10^{-2}	
7.5 x 10^{-2}	

Table 3.9: Additional hyperparameters.

Hyperparameter	**Value**
Train set size	76.5 %
Validation set size	13.5 %
Test set size	10.0 %
Batch size	32
Regularization	None
Number of epochs	15

3. Additional Hyperparameters

The remaining hyperparameters were tested manually using the selected result from Step 2. Depending on the evaluation results, hyperparameters were changed, potentially leading to better results. Similarly to prior steps, the goal was to improve the validation accuracy as much as possible.

- ☐ **Train, test and validation set size**
- ☑ Optimization technique
- ☑ Loss function
- ☑ Learning rate
- ☐ **Regularization techniques**
- ☐ **Number of epochs**
- ☐ **Batch size**
- ☑ Model configuration
  - ☑ Activations
  - ☑ Kernel parameters
  - ☑ Pooling parameters

The items on the following list were tested:

1. Batch size: 16, 64 and 128.
2. Train/val/test ratio in percent: 10/9/81, 10/18/72.
3. L2 regularization: weight decay = 1e-2.

4. Final Run and Live-Classification

As a last step, the best resulting hyperparameter setup from all the hyperparameter steps was selected for the final training of the model. Due to the properties of the training configuration, there were random elements varying with each run of the model, leading to slightly different results each time. Accordingly, the same configuration was run three times. The resulting model parameters achieving the maximum validation accuracy were tested using the testing set, before they were saved in order to be used for live-classification. The live-classification was simulated using unseen data. A forward pass through the network was applied every 30 s for each 1.5 km long overlapping segment along the entire 51 km long railway line section.

3.6.2 Performance Evaluation

The final evaluation of the model and its parameters was based on a confusion matrix, as well as manual inspection of predictions. These evaluation methods are both well suited for multi-class problems with image data.

CHAPTER 4

RESULTS

This chapter presents all relevant results achieved during this research. Similarly to Section 3.6 in Methods, this chapter follows the same four steps, presenting the results from each step in chronological order. Additionally, results from the final run of the model on the testing set, along with evaluation of this final run and a simulation of live-classification is presented.

1. **Iterate through models**
2. **Iterate through learning rates and optimizers**
3. **Investigate additional hyperparameters**
4. **Final run and live-classification**

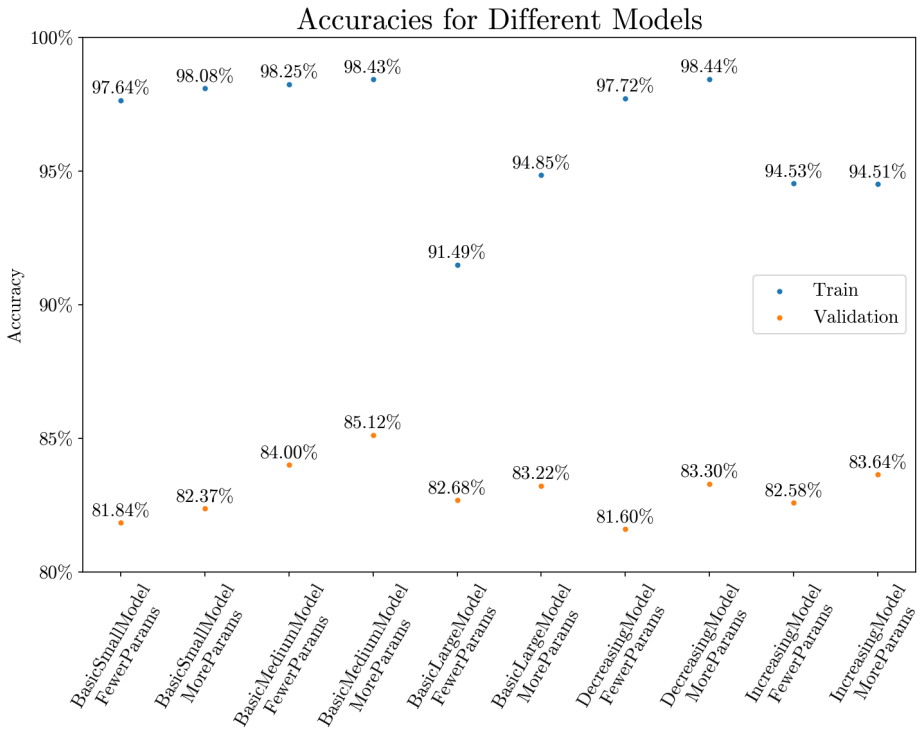

Figure 4.1: Training and validation accuracies for the different CNN variants.

1. Model Iteration

In the following, the results from Step 1 of the hyperparameter tuning in Methods Section 3.6.1 are presented. This includes Figure 4.1 displaying training and validation accuracies for the different models and Figure 4.2 displaying the learning curves for two selected models, BasicMediumModelMoreParams (BMMMP) and BasicLargeModelFewerParams (BLMFP).

Figure 4.1 displays the relationship between training and validation accuracies acquired in Step 1. The accuracies displayed for each model on the x-axis are the maximum validation accuracies acquired through 15 epochs, along with their corresponding training accuracy from the same epoch. The highest validation accuracy belongs to BMMMP at 85.12 %. The closest relationship between training and validation accuracies belongs to BLMFP, which also inhabits the lowest training accuracy.

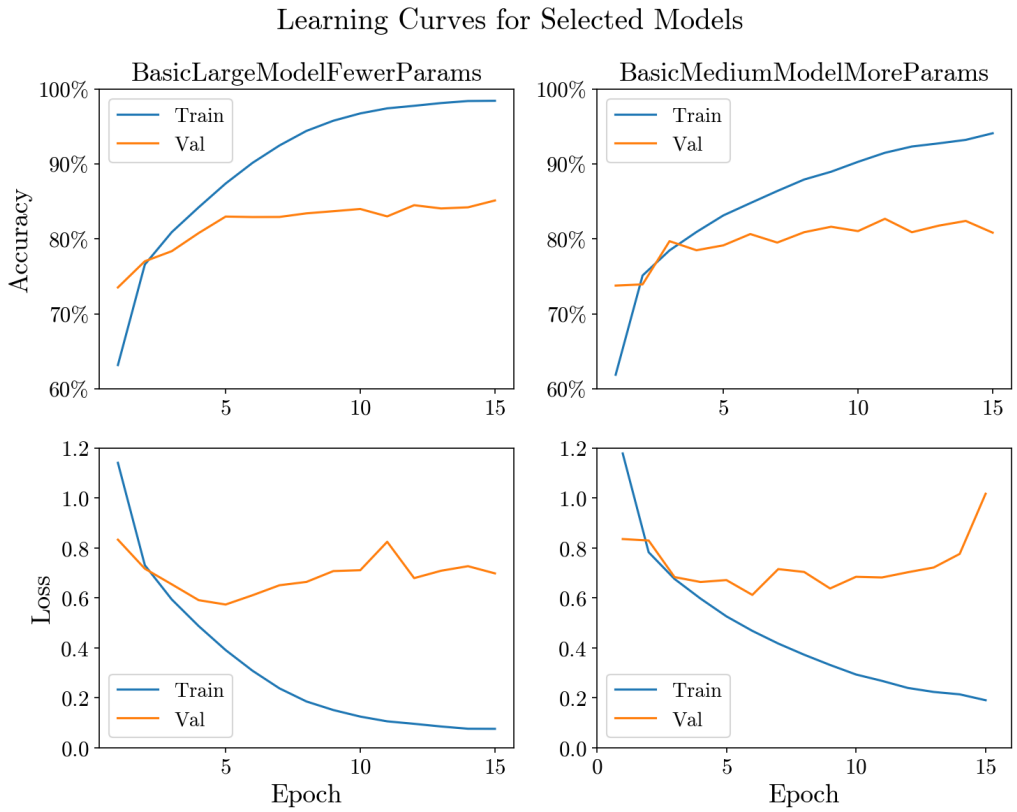

Figure 4.2: Learning curves for the two selected models.

Figure 4.2 displays the training and validation accuracy curves for BMMMP and BLMFP. Similar for both curves is an initial higher validation accuracy than training accuracy with only minor improvements. Due to the properties of the training and validation loops, the model updates its initial parameters before calculating validation metrics for each epoch, which might explain why the validation metrics are initially better than training metrics. Additionally, see Appendix B for the model training and validation script. The convergence point is at around three epochs for both models, marking the point for further overfitting. See Appendix C for learning curves of all models. Additionally, see Appendix D for a plot displaying more "classic" learning curves (with lower learning rate).

2. Learning Rates and Optimizer Iteration

In the following, the results from Step 2 of the hyperparameter tuning are presented. Two models from Step 1 are reviewed, BMMMP and BLMFP. Table 4.1 lists the maximum validation accuracies for each model and optimizer, along with their corresponding training accuracies and learning rates. Figure 4.3 displays the maximum validation accuracies for BMMMP as a function of learning rate for different optimizers. Lastly, the corresponding learning curves for BMMMP are displayed in Figure 4.3.

Table 4.1: Presentation of the highest validation accuracies for each optimizer for the two models in Step 2, along with the corresponding training accuracies and learning rates. The maximum validation accuracy for BMMMP is marked with bold text.

	BasicLargeModelFewerParams				**BasicMediumModelMoreParams**			
Optimizer	**SGD**	**RMSprop**	**Adagrad**	**Adam**	**SGD**	**RMSprop**	**Adagrad**	**Adam**
Val. acc.	82.74 %	83.65 %	81.63 %	83.00 %	85.04 %	**85.68 %**	83.69 %	84.59 %
Tr. acc.	97.11 %	97.45 %	88.98 %	96.27 %	98.68 %	98.58 %	96.63 %	98.50 %
LR	0.025	0.0001	0.005	0.0001	0.025	0.0001	0.001	0.0001

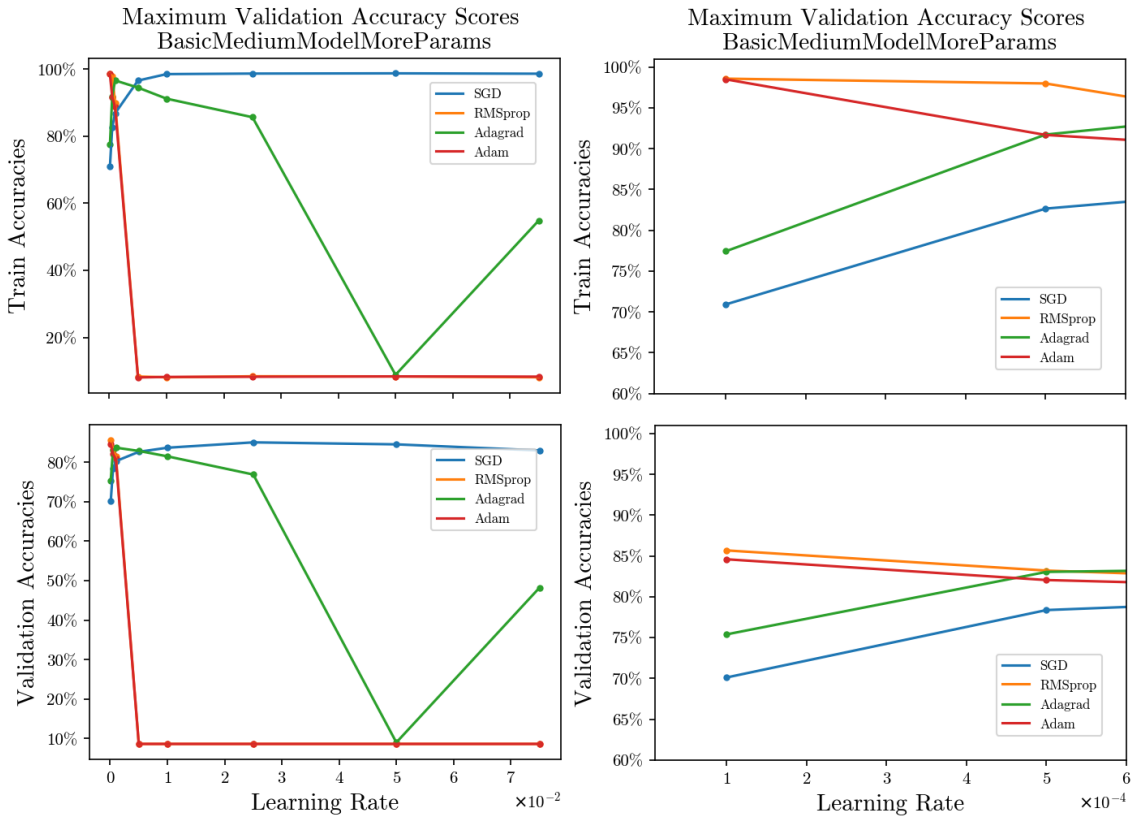

Figure 4.3: Validation accuracies (bottom) and their corresponding training accuracies (top) as functions of learning rater for different optimizers (see legends) for the BasicMediumModelMoreParams model. The validation accuracies were selected as the maximum achieved throughout 15 epochs. The right column shows a closeup of the lower learning rates.

Figure 4.3 displays the maximum validation accuracies and the corresponding training accuracies out of 15 epochs for BMMMP as a function of learning rate for different optimizers (see legend). See Appendix C for a similar plot of BLMFP. The two leftmost plots display all iterations, while the rightmost are zoomed in on the lower learning rates. The highest validation accuracy belongs to RMSprop on the lowest learning rate of 0.0001 (see Table 4.1). Note that the RMSprop curve is barely visible behind the Adam curve for the higher learning rates. The SGD curve has higher accuracies on the higher learning rates. Adam and RMSprop have higher accuracies on lower learning rates, before dropping down to around 8 % on higher learning rates. This fast drop in accuracy for the different optimizers on the higher learning rates might be an overshoot effect, where the learning rate is so large that it overshoots the local minimum, leading to an increased loss.

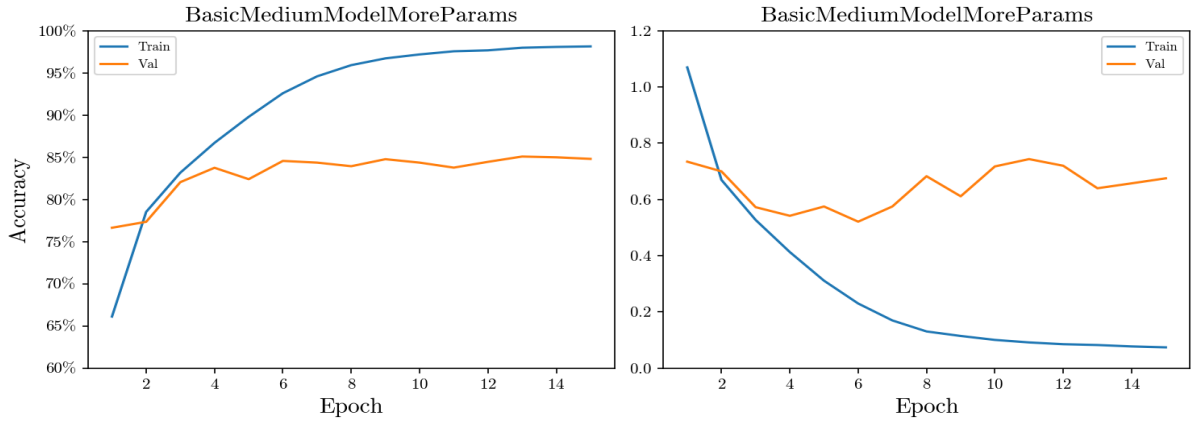

Figure 4.4: Learning curves for BMMMP.

Figure 4.4 illustrates the learning curves for BMMMP using RMSprop and a learning rate of 0.0001. The convergence point is around four epochs, at which point the model starts to overfit to the training data. The general shape of the validation curves show similar characteristics as the curves in Step 1 (see Figure 4.2).

3. Additional Hyperparameters

Following, the results from Step 3 of the hyperparameter tuning process are presented. Based on the resulting hyperparameter settings from step 2 (RMSprop optimizer, learning rate 0.0001, batch size 32, 15 epochs and a training/validation/testing ratio of 76.5/13.5/10 % respectively), additional settings were tested, according to the list given in Section 3.6.1 (hyperparameter values also given in Table 4.2).

Table 4.2: Additional hyperparameter testing results. WD stands for weight decay, while Ratio is the training/validation/testing set sizes in %.

	Batch Size			**Regularization**	**Ratio**	
	16	64	128	WD(1e-2)	9/81	18/72
Val. acc.	84.7 %	78.28 %	84.19 %	76.89 %	85.59 %	85.30 %
Tr. acc.	98.2 %	84.54 %	97.16 %	77.48 %	97.64 %	98.14 %

4. Final Run and Live-Classification

The hyperparameter setup achieving the highest validation accuracy in the three previous steps of the hyperparameter tuning was acquired in Step 2. Accordingly, Table 4.3 lists all hyperparameters used for the final run of the model. See Appendix A for the BMMMP architecture in Python.

Table 4.3: Additional hyperparameters applied.

Hyperparameter	**Value**
Model	BMMMP
Loss function	Cross-entropy loss
Train set size	76.5 %
Validation set size	13.5 %
Test set size	10.0 %
Optimizer	RMSprop
Learning rate	0.0001
Batch size	32
Regularization	None
Number of epochs	15

For the final run, the testing dataset was passed forward through the final model configuration, resulting in a test loss of 0.7730 and a general test accuracy of 84.98 %. See Table 4.4 for specific class accuracies on the test set. Additionally, the result from the simulated live-prediction is presented in Figure 4.5. The predicted class is plotted on top of the corresponding signal region.

Table 4.4: Specific class testing accuracies.

Class index	**0**	**1**	**2**	**3**	**4**	**5**
Testing accuracy	75.05 %	72.41 %	94.46 %	92.51 %	92.81 %	92.95 %
Class index	**6**	**7**	**8**	**9**	**10**	**11**
Testing accuracy	97.21 %	98.13 %	78.78 %	83.23 %	70.65 %	72.62 %

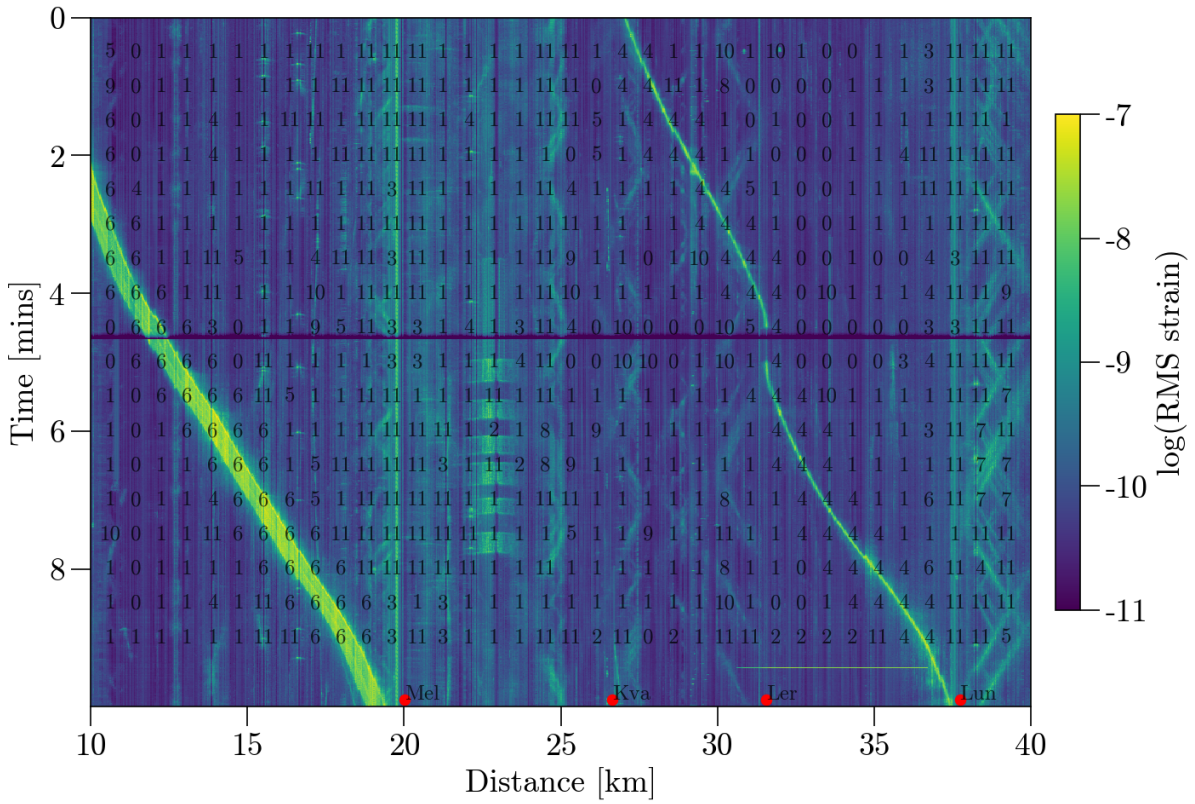

Figure 4.5: Live-classification results: The predicted class labels overlaying RMS DAS data that has not been previously seen by the model. The red dots denote the train stations.

4.1 Classification Results and Evaluation

4.1.1 Confusion Matrix

In Figure 4.6, the computed confusion matrix from the final run on the testing set is presented. The numbers in each window corresponds to the testing accuracy as decimal fractions of 1 (corresponding to % if multiplied by 100). The accuracies are generally high for the train classes and the stationary and artifact noise class. However, notice the confusion between the vehicle classes, especially in the class with both directions. Adding together all the predicted vehicle labels for "car NS SN" results in a total accuracy of 89 % (car SN: 96 %, car NS: 89 %). Moreover, the remaining classes (ambient and unknown noise and multiple objects) all display similar characteristics, where the true class was falsely predicted as other classes, seemingly at an even rate for almost all other classes. Besides, notice that many of the classes have high prediction scores for unknown noise.

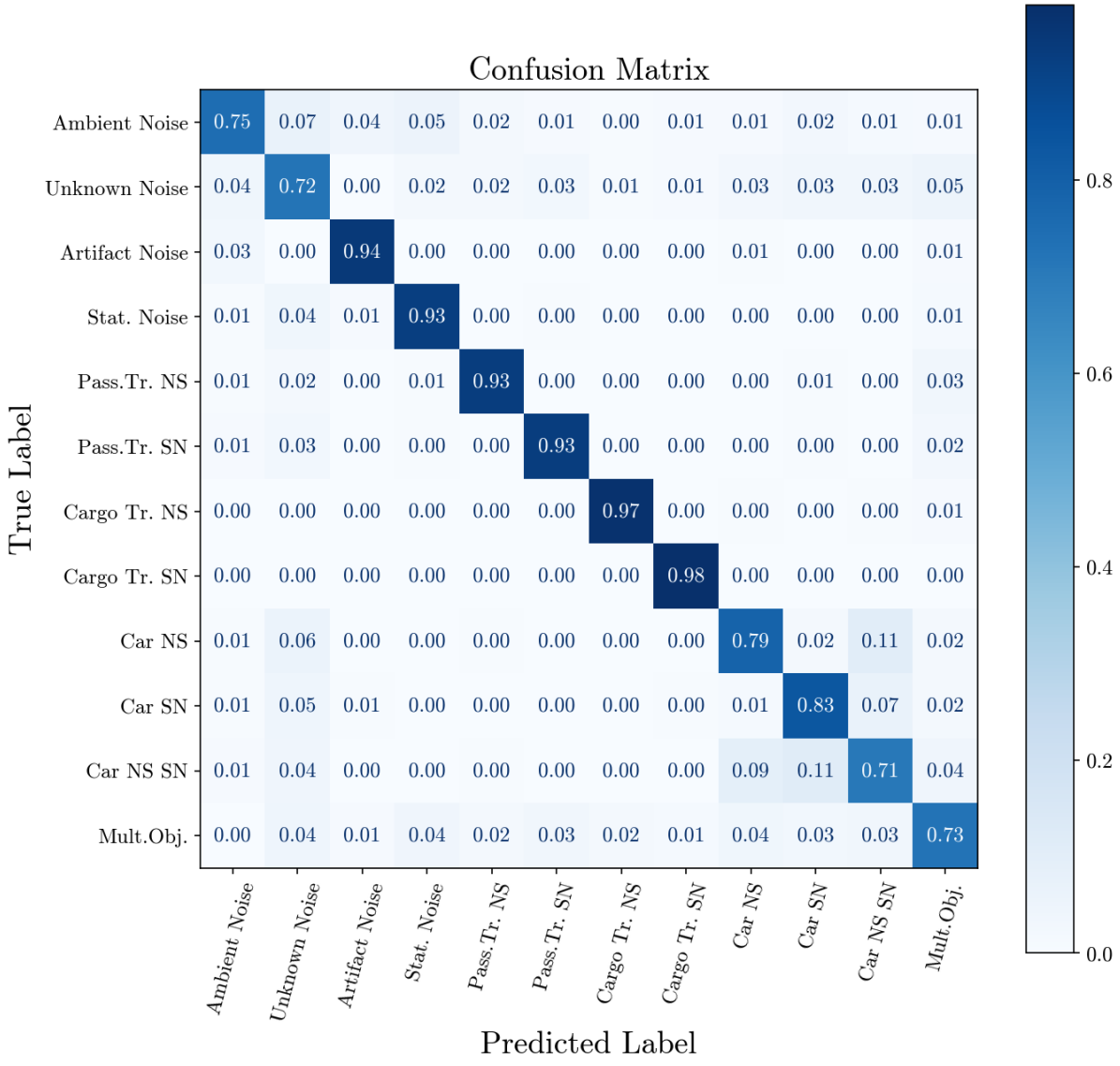

Figure 4.6: Confusion matrix from final model run

4.1.2 Manual Inspection of Incorrect Predictions

In this section, a figure displaying incorrect predictions on the test set is presented. The legend in each subplot indicates the true and predicted labels. See Table 3.1 for a mapping between class names and indexes. From Figure 4.7, it is visible that the "lower right" window has an incorrect label. This window, however, is displayed as correctly predicted by the model. The true label of the "left center" window is multiple objects (which contains train stop windows), while the model predicted it to be cargo train class. Moreover, "upper center" and "right center" both display "edge cases" of the ambient noise class, however, the model predicted the signal displayed in the corner of the image. Moreover, the incorrect vehicle predictions "lower left", "true center", "upper right" and "upper left" all display predictions of a different vehicle class. See Appendix E for additional figures of incorrect predictions.

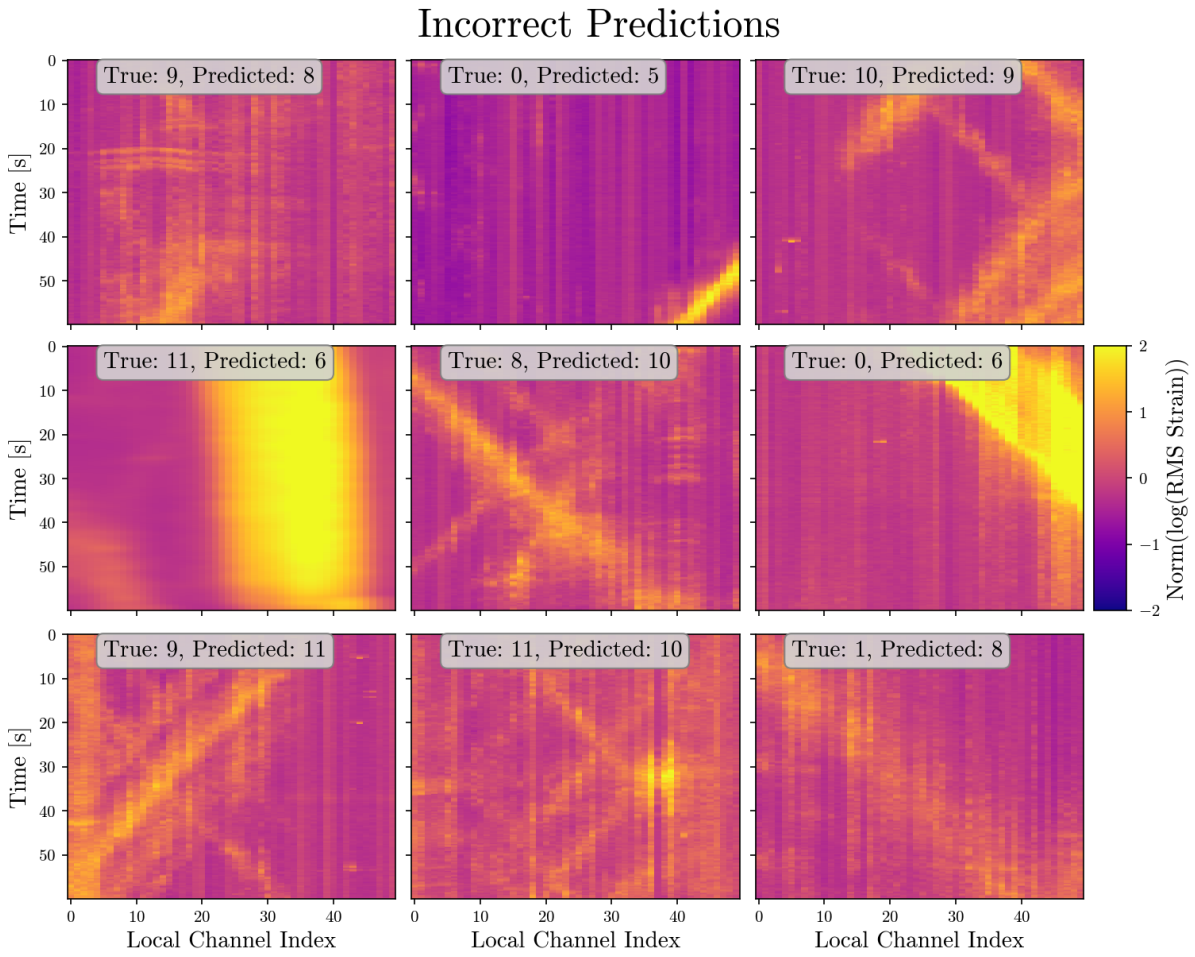

Figure 4.7: Exemplary windows with incorrect predictions

CHAPTER 5

DISCUSSION

In order to assess the feasibility of integrating DAS and CNNs for railway monitoring, the aim of this research was to create a framework for a live-monitoring system of railway DAS signals, by the use of a CNN. This chapter discusses both whether the goals are met and the potential implications of the results in-depth. Additionally, future work recommendations are given based on the discussed limitations.

A process of four steps was utilized in order to create a CNN for classification of DAS railway signals and tune its hyperparameters. The dataset was manually labeled and augmented using custom scripts. The four steps of the hyperparameter tuning resulted in increasingly better general metrics (validation accuracies and losses), before a final run with optimal hyperparameters was performed. This final run resulted in a general testing accuracy of 84.98 %, with specific class accuracies ranging from 70.65 % to 98.13 %. Manual inspection of the incorrect predictions indicate labeling (or augmentation) mistakes, along with edge-case mistakes in the dataset as being the main error types in the test set prediction. The live-classification resulted in seemingly good results, indicating that the method has potential for use as a live-monitoring tool by train companies.

5.1 Result Interpretations

Generally, the hyperparameter tuning did not lead to significantly higher validation accuracies in either of the steps. With the initial lowest validation accuracy being 81.60 % in Step 1, the increase to 85.68 % was the highest validation accuracy acquired during the tuning. This is an increase of only 4.08 %. This marginal progress prompts a closer examination of potential limitations in both the dataset and the model tuning.

5.1.1 Class Diversity vs. Accuracy

The separate classification accuracies on the test set displayed in Figure 4.6 and Table 4.4 reveal a potential correlation between class accuracies and class diversity. As displayed, the results reveal lower specific testing accuracies in among others the unknown noise class and the multiple objects class. Both of these classes inherent high diversity (ie. lots of variations within the class). See Figure 3.5 and 3.4 for labeled and unlabeled windows, respectively. On the other hand, for instance the signal type in one of the train classes is more heavily represented in the entire dataset, as opposed to the many different signal types in the multiple objects class. This is consistent with the results in the confusion matrix and suggests that the model struggles more with classes that have higher diversity.

This makes sense, as the model would need to learn more patterns and relationships in these classes than the first (ie. there is dataset bias [46]). As a solution, simply increasing the dataset size may not necessarily resolve the problem, as it would not decrease the bias. However, a potential solution could be to apply augmentation techniques in such a way that the resulting class diversities are evened out. By doing this to the lower diversity classes, the diversity would be increased. A second solution could be to divide the classes which inherent more diversity into additional sub-classes, effectively reducing the diversity in each class. Moreover, an additional method could be researched, where two models are trained for the lower and higher diversity classes separately [46].

5.1.2 Vehicle Classes

Further analysis of the confusion matrix in Figure 4.6 reveals inconsistencies and confusion between the three vehicle classes, and in particular the bidirectional vehicle class (car NS SN). This revelation indicates a potential issue with consistency in the labeling process due to challenges with differentiating between slight differences in amplitude in a consequent manner. This issue is further verified by analyzing the plotted incorrect predictions in Figure 4.7, where several of the subplots presents this case. As a potential solution, merging individual car classes into a unified class for all vehicles in all directions could enhance the vehicle prediction accuracy. However, this solution would require careful consideration, as the merging process would introduce increased diversity within the unified class, potentially posing new challenges. Besides, the decision should align with the end user's specific needs and whether the primary goal is to detect the vehicle in any direction or in a specified direction.

5.1.3 Incorrect Class Labels in Dataset

By analyzing the incorrect prediction results in Figure 4.7 and Appendix E, it is apparent that a number of the images are incorrectly labeled. This would not be a big issue if those were only a few of the augmented windows. However, if the mistake is made prior to augmentation, one incorrect image could result in a total of 72 incorrect images, which might be a bigger problem. Nevertheless, the model appears to make the correct prediction despite the incorrect labels. This suggests the presence of a sufficient amount of correctly labeled data. This method of evaluation (analysis of incorrectly predicted images) should, however, be used with caution, since only a small selection of predictions is presented and reviewed. This small selection of results may not be representative for the entire set of results, which may lead to imprecise interpretations and conclusions. Accordingly, it is a good practice having this in mind when making assumptions based on only a small sample size of results.

Still, a definitive conclusion that can be drawn from this evaluation method, is the fact that there exist incorrect labels in the dataset. The exact magnitude of these, however, remains challenging to quantify. The presence of incorrect labels emphasizes the potential benefit from a revision of both the labeling and augmentation process. By implementing additional checks during both the labeling and augmentation process, or by using only the windows flagged with "yes" for augmentation compatibility for the augmentation process, the amount of incorrect labels could be reduced. Moreover, the model developed in this research may in itself play a role in identifying the mislabeled windows. By integrating a mechanism in the model to map the incorrect predictions to their unique ID and additionally flag them with their predicted class label, these flags can then be manually inspected. This could lead to a further reduction of incorrect labels in the dataset.

5.2 Implications

The results from both the test set confusion matrix and the simulated real-time classification results indicate that this tool may already be successful if used live as a train and vehicle monitoring tool along railway lines. However, it should be approached with caution if the tool is to be utilized for tracking the classes with lower testing accuracies, like monitoring of a specific vehicle direction (NS, SN or NS SN). Additionally, as there is generally low chances of the model classifying a real event as ambient noise (see predicted label for ambient noise in confusion matrix), the tool might be successfully applied for event detection. However, there would be a chance of false detection since the ambient noise accuracy is 75.05 %.

Moreover, the model developed in this research may already be used for detection of unknown events (unknown noise) for low stake applications. By using the tool in combination with a human operator, unknown events and other events can be detected and further investigated. Additionally, the methods developed are adaptable, making them applicable for utilization (and further development) in locations with similar characteristics (railway lines). The results from this research suggest that the integration of DAS and a CNN has the potential to be a cost-effective monitoring tool for railway events, which could have direct benefits for railway operators (like BaneNOR). The research outcome additionally opens up an opportunity of further exploiting additional fibers co-located with railway lines, enabling potential automatization of real-time railway traffic monitoring. However, the reliability of the developed tool for practical use is still not ideal, requiring additional research on potential refinements.

5.3 Methods Assessment and Limitations

The generalizability of the results in this research is limited by utilizing data collected within a limited area and time frame. Hence, by incorporating additional (similar) datasets to the existing one, the tool might become more generalizable. Moreover, the dataset used in this research completely lacks the "cargo train SN" class. For this research, this entire class consists of augmented data, where the original data is sampled from "cargo train NS". Whether this causes significant issues or not is difficult to say without applying the model to testing data consisting of this class. Despite of lacking examples in the "cargo train SN" class, the specific time frame was chosen for the dataset since there were significantly less gaps (caused by the IU) present in the data compared to other times. Gaps disrupt the real signals and create signals that cannot be naturally present in the data and might thus distort the classifier. However, the magnitude of this issue is unknown.

Lastly, the dataset does not provide definitive ground truths, as the labels are based on what is visually inspected on the RMS strain signals. Passenger trains were verified using the train schedule and the localized train stations. Additionally, several train and car signals were logged during the data acquisition. However, the lack of ground truths is especially true for the unknown noise class. Optionally, listening to the signals might give confirmation or additional information to the visual information which is already explored. Moreover, a different alternative could be setting up cameras along the fiber, and if synchronized with the strain signal for a certain period of time, it can act as a mapping for the window labeling process. The mapping from camera feed to data labels could even be automized using a well established object detection method from computer vision like YOLO [47]. In addition to the benefit of obtaining ground truths, this could significantly reduce the manual labor required during the labeling process.

Within this work, only one type of data normalization was applied. Other methods, like global contrast normalization, which is a common normalization method for image classification [48], or batch normalization, which is a popular technique for CNNs [49], could be beneficial to try. Especially batch normalization, as it can be argued to enhance the performance and stability of neural networks [50].

During the testing of hyperparameters, assessing the optimal settings from a large number of potential values and their combinations poses a challenge. Attempting to test all possible combinations would likely be the most certain way to achieve the best performance possible, but it would be extremely time-consuming and thus considered unfeasible. Nevertheless, doing more research on how to tune a model in the best way would increase the likelihood for better performance.

By assessing the four step method utilized in this research, several potential flaws can be found. For instance, there is the order of which hyperparameters were tuned first to last. By stepwise eliminating some hyperparameter values based on performance in the current step, better-performing combinations in the next step could potentially get lost in the process. Using nested loops for iterating through combinations of hyperparameters (step 2) would help, but in order to test every combination, one would require extreme amounts of processing, as the dimensionality would increase in the fold of number of tested values per added loop. An additional flaw is the number of training runs performed in each tuning step. With exception of the last step, the trainings in this research were only run once per step. Due to the randomized elements introduced in the training process (randomized weight initialization and dataset shuffling), performing only one run might have led to non-optimal selections of hyperparameter values. Moreover, out of all existing hyperparameters and values, the limited number of them tested in this research may not have been the most optimal for the given dataset.

Additionally, the evaluation performed after each step of the tuning process may not have been the most optimal. It was mainly based on maximum validation accuracy, which may not have been the most accurate evaluation metric. By analyzing the resulting learning curves from each step in the tuning process, it is evident that the model is overfitting to the data. Despite this, the model performance on the test set did not seem to be significantly affected, achieving almost as good accuracy as the validation set. In hindsight, evaluation based more heavily on the patterns of learning curves may have been more successful in the attempt of finding the most optimal hyperparameter values (see 2.5.1). This would include paying attention to the level of overfitting through the entire process as well as for the final run, which might have resulted in overall better performance.

An additional aspect worth mentioning is the fact that for this research, the effect of batch size on model performance does not fit with the theory given in Section 2.5. According to this theory, having smaller batches (or a higher batch size) should increase model performance. Despite this, the optimal batch size in this research was found to be 32 (tested up to 128). However, according to several sources, the negative effect of having too large batches might be counteracted by increasing the learning rate (or by not decaying the learning rate [51]) [38]. Looking at Figure 3 in Appendix D, it is visible that a lowered learning rate on a batch size of 32 decreases the accuracy, which strengthens this hypothesis (ie. opposite way of increasing learning rate on a large batch). Nevertheless, more research should be performed in order to make a certain conclusion.

5.3.1 Future Work

It is beyond the scope of this research to provide a fully developed railway monitoring-tool. Instead, this research aim was to develop a framework, or a general method, of doing so, and to assess its feasibility for further development. Even though railway companies might already benefit from the developed monitoring tool, the results and discussion also raises questions about certain aspects of the tool, meaning it could benefit from further research. Based on all the work done in this research, recommendations for further development of the method are proposed.

5.3.1.1 Dataset Development

The varying accuracies among classes highlight the need for dataset refinement. Exploring additional augmentation techniques and collecting more diverse training data, possibly from different locations or time periods, can enhance the model's generalizability.

Augmentation strategies may benefit from revision, by using them in such a way that the image diversities across classes are equalized. Similarly, a re-evaluation of classes, (either by splitting classes into sub-classes, or by merging others into one class) may benefit the model training, resulting in higher accuracies.

The labeling process which was based on personal assessment, introduced uncertainties, particularly in classes with multiple objects which can only be differentiated by slight differences in amplitude. More emphasis should be put into developing effective labeling "rules" in order to ensure correct labels. Additionally, with some modifications, this classifier may be used as a tool to re-label incorrect labels based on the prediction results. (In that case the window IDs need to be mapped with the predicted windows.)

Moreover, additional preprocessing steps may be considered, for instance different normalization techniques. Additionally, in order to increase the generalizability of the model, collection of more data from other locations and periods of time should be considered.

5.3.1.2 Modeling and Training

Further research is needed to establish an optimal hyperparameter tuning method. The four iterative tuning steps in this work may benefit from revision. For instance, adding more hyperparameter testing, changing the order of the tests or expanding the tested value range for each hyperparameter.

Additional research on the model should be considered, for instance exploring deepening the fully connected layers, adding dropout, trying other activation functions or adding weights based on certainties from the labeling process.

CHAPTER 6

CONCLUSIONS

The aim of this research was to provide a framework for a live-monitoring system of railway DAS signals by the use of a CNN. The resulting general testing accuracy of 84.98 %, in addition to the simulated live-classification of the CNN, suggest that DAS and CNN integration is a viable option for railway signal classification. Moreover, extensive hyperparameter tests have been conducted to optimize the learning process of the network and improve its accuracy. Despite of promising results, future work on reliability will be necessary for the practical implementation of the monitoring tool. The large variety in specific class accuracies ranging from 70.65 % to 98.13 % is an indication that further work on both the dataset as well as the model is needed. However, the created framework may already be utilized for low stake applications. With further improvements, the findings of this research can open up an opportunity for a large scale exploitation of fibers co-located with railway lines, potentially enabling automatization of real-time railway traffic monitoring. Lastly, due to the inherent adaptability of the framework, the results of this research may be beneficial for other research and applications even beyond the field of DAS and geophysics.

REFERENCES

[1] Alexander S. "Tesla's Use of AI: A Revolutionary Approach to Car Technology". In: (2023). URL: https://www.linkedin.com/pulse/teslas-use-ai-revolutionary-approach-car-technology-alexander-stahl/.

[2] "10 LATEST DEVELOPMENTS IN ARTIFICIAL INTELLIGENCE". In: (2023). URL: https://moonpreneur.com/blog/latest-developments-in-artificial-intelligence/.

[3] TechInsights Hub. "Distributed Acoustic Sensing (DAS) Market | Expected To Be the Fastest Growing Industry 2036". In: (2023). URL: https://www.linkedin.com/pulse/distributed-acoustic-sensing-das-market-expected-fastest-evaof/.

[4] "Global Distributed Acoustic Sensing Market Insights". In: (2024). URL: https://www.skyquestt.com/report/distributed-acoustic-sensing-market#:~:text=Global%20Distributed%20Acoustic%20Sensing%20Market%20Insights,period%20(2023%2D2030)..

[5] Ceyhun Efe Kayan, Kivilcim Yuksel Aldogan, and Abdurrahman Gumus. "An Intensity and Phase Stacked Analysis of Phase-OTDR System using Deep Transfer Learning and Recurrent Neural Networks". In: (2022). DOI: 10.1364/AO.481757. URL: https://arxiv.org/abs/2206.12484.

[6] Zhaoqiang Peng et al. "Identifications and classifications of human locomotion using Rayleigh-enhanced distributed fiber acoustic sensors with deep neural networks". In: (2020). DOI: 10.1038/s41598-020-77147-2. URL: https://www.nature.com/articles/s41598-020-77147-2.

[7] Iñigo Corera et al. "Long-Range Traffic Monitoring Based on Pulse-Compression Distributed Acoustic Sensing and Advanced Vehicle Tracking and Classification Algorithm". In: (2023). DOI: 10.3390/s23063127. URL: https://www.mdpi.com/1424-8220/23/6/3127.

[8] Avinash Nayak and Jonathan Ajo-Franklin. "Distributed Acoustic Sensing Using Dark Fiber for Array Detection of Regional Earthquakes". In: (2021). DOI: 10.1785/0220200416. URL: https://pubs.geoscienceworld.org/ssa/srl/article-abstract/92/4/2441/595405/Distributed-Acoustic-Sensing-Using-Dark-Fiber-for?redirectedFrom=fulltext.

[9] Pablo D. Hernández, Jaime A. Ramírez, and Marcelo A. Soto. "Deep-Learning-Based Earthquake Detection for Fiber-Optic Distributed Acoustic Sensing". In: (2022). URL: https://opg.optica.org/jlt/abstract.cfm?uri=jlt-40-8-2639.

[10] Huan Wu et al. "Pattern recognition in distributed fiber-optic acoustic sensor using an intensity and phase stacked convolutional neural network with data augmentation". In: (2021). DOI: 10.1364/OE.416537. URL: https://opg.optica.org/oe/fulltext.cfm?uri=oe-29-3-3269&id=446729.

[11] Md Arifur Rahman, Suhaima Jamal, and Hossein Taheri. "Remote Condition Monitoring of Rail tracks using Distributed Acoustic Sensing (DAS): A Deep CNN-LSTM-SW based Model". In: (2024). DOI: 10.1016/j.geits.2024.100178. URL: https://www.sciencedirect.com/science/article/pii/S2773153724000306.

[12] Shulun Wang, Feng Liu, and Bin Liu. "Research on application of deep convolutional network in high-speed railway track inspection based on distributed fiber acoustic sensing". In: (2021). DOI: 10.1016/j.optcom.2021.126981. URL: https://www.sciencedirect.com/science/article/pii/S0030401821002315.

[13] K.A. Gerardino. "DIGITALIZATION IN THE RAILWAY INDUSTRY". In: (2022). URL: https://railway-international.com/market-overview/62652-digitalization-in-the-railway-industry.

[14] Anna Sophie Nymoen Tveit. "everaging Unsupervised Machine Learning Methods for Distributed Acoustic Sensin". In: (2023).

[15] Arthur Hartog. *An Introduction to Distributed Optical Fibre Sensors*. 1st edition. June 2017. DOI: 10.1201/9781315119014.

[16] Yang Zhi, Shi Pengxiang, and Li Yongqian. "*Research on COTDR for measuring distributed temperature and strain*". In: (2011). DOI: 10.1109/MACE.2011.5986993.

[17] W. Lowrie and A. Ficthner. *Fundamentals of Geophysics*. 3rd edition. Cambridge University Press, Mar. 2020, pp. 423–428.

[18] Tom Parker, Sergey Shatalin, and Mahmoud Farhadiroushan. "*Distributed Acoustic Sensing - A new tool for seismic applications*". In: *First Break* 32 (Feb. 2014). DOI: 10.3997/1365-2397.2013034.

[19] Kittinat Taweesintananon et al. "*Distributed acoustic sensing for near-surface imaging using submarine telecommunication cable: A case study in the Trondheimsfjord, Norway*". In: 86 (Sept. 2021). DOI: 10.1190/geo2020-0834.1.

[20] Michael Copeland. "What's the Difference Between Artificial Intelligence, Machine Learning and Deep Learning?" In: (2016). URL: https://blogs.nvidia.com/blog/whats-difference-artificial-intelligence-machine-learning-deep-learning-ai/.

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

[22] Pragati Baheti. "Activation Functions in Neural Networks [12 Types & Use Cases]". In: (2021). URL: https://www.v7labs.com/blog/neural-networks-activation-functions#:~:text=The%20linear%20activation%20function%2C%20also,the%20value%20it%20was%20given..

[23] Ryan Holbrook and Alexis Cook. "Convolution and ReLU". In: (2023). URL: https://www.kaggle.com/code/ryanholbrook/convolution-and-relu.

[24] Wang Hao et al. "The Role of Activation Function in CNN". In: (2020).

[25] Björn Lindqvist et al. “Chapter 12 - ARW deployment for subterranean environments”. In: *Aerial Robotic Workers*. Ed. by George Nikolakopoulos, Sina Sharif Mansouri, and Christoforos Kanellakis. Butterworth-Heinemann, 2023. ISBN: 978-0-12-814909-6. DOI: https://doi.org/10.1016/B978-0-12-814909-6.00018-4. URL: https://www.sciencedirect.com/science/article/pii/B9780128149096000184.

[26] Andrej Karpathy. “CS231n Winter 2016: Lecture 7: Convolutional Neural Networks”. In: (2016). URL: https://www.youtube.com/watch?v=LxfUGhug-iQ.

[27] Lauren Holzbauer. “Convolutional Neural Networks Explained...with American Ninja Warrior”. In: (2019). URL: https://blog.insightdatascience.com/convolutional-neural-networks-explained-with-american-ninja-warrior-c6649875861c.

[28] Sagar Sharma. “Activation Functions in Neural Networks”. In: (2017). URL: https://towardsdatascience.com/activation-functions-neural-networks-1cbd9f8d91d6.

[29] Zhou and Chellappa. “Computation of optical flow using a neural network”. In: *IEEE 1988 International Conference on Neural Networks*. 1988. DOI: 10.1109/ICNN.1988.23914.

[30] Ivan B. Djordjevic. “Chapter 14 - Quantum Machine Learning”. In: *Quantum Information Processing, Quantum Computing, and Quantum Error Correction (Second Edition)*. Ed. by Ivan B. Djordjevic. Second Edition. Academic Press, 2021. ISBN: 978-0-12-821982-9. DOI: https://doi.org/10.1016/B978-0-12-821982-9.00007-1. URL: https://www.sciencedirect.com/science/article/pii/B9780128219829000071.

[31] Minhajul Hoque. “Demystifying Neural Network Normalization Techniques”. In: (2023). URL: https://medium.com/@minh.hoque/demystifying-neural-network-normalization-techniques-4a21d35b14f8#:~:text=This%20can%20help%20to%20prevent,in%20the%20input%20or%20weights..

[32] Tarang Shah. “About Train, Validation and Test Sets in Machine Learning”. In: (2017). URL: https://towardsdatascience.com/train-validation-and-test-sets-72cb40cba9e7.

[33] Kizito Nyuytiymbiy. “Learning”. In: (2020). URL: https://towardsdatascience.com/parameters-and-hyperparameters-aa609601a9ac.

[34] Jason Brownlee. “How to use Learning Curves to Diagnose Machine Learning Model Performance”. In: (2019). URL: https://machinelearningmastery.com/learning-curves-for-diagnosing-machine-learning-model-performance/.

[35] Dipam Vasani. “This thing called Weight Decay”. In: (2019). URL: https://towardsdatascience.com/this-thing-called-weight-decay-a7cd4bcfccab.

[36] Yash Agrawal. “The Underlying Dangers Behind Large Batch Training Schemes”. In: 2022.

[37] Nitish Shirish Keskar et al. “On Large-Batch Training for Deep Learning: Generalization Gap and Sharp Minima”. In: (2017). DOI: 10.48550/arXiv.1609.04836. URL: https://arxiv.org/abs/1609.04836.

[38] Kevin Shen. “Effect of batch size on training dynamics”. In: 2018.

[39] Gregory Naitzat, Andrey Zhitnikov, and Lek-Heng Lim. "Topology of Deep Neural Networks". In: *Journal of Machine Learning Research 21 (2020) 1-40* (2020). URL: https://jmlr.csail.mit.edu/papers/volume21/20-345/20-345.pdf.

[40] David E. Rumelhart, Geoffrey E. Hinton, and Ronald J. Williams. "Learning representations by back-propagating errors". In: *Nature* 323 (1986), pp. 533–536. URL: https://api.semanticscholar.org/CorpusID:205001834.

[41] Claudia Perlith. "Learning Curves in Machine Learning". In: (2011). DOI: 10.1007/978-0-387-30164-8_452. URL: https://dominoweb.draco.res.ibm.com/reports/rc24756.pdf.

[42] "CS231n Convolutional Neural Networks for Visual Recognition". In: (). URL: https://cs231n.github.io/neural-networks-3/#sanitycheck.

[43] T. S. Hudson et al. "*Distributed Acoustic Sensing (DAS) for Natural Microseismicity Studies: A Case Study From Antarctica*". In: *Journal of Geophysical Research: Solid Earth* 126.7 (2021). DOI: 10.1029/2020JB021493.

[44] MJ Cunningham and GL Bibby. *11 - Electrical Measurement*. Ed. by M.A. Laughton and D.J. Warne. Sixteenth Edition. Oxford: Newnes, 2003. DOI: 10.1016/B978-075064637-6/50011-3.

[45] Min Lin, Qiang Chen, and Shuicheng Yan. "Network In Network". In: (2014). DOI: 10.48550/arXiv.1312.4400. URL: https://arxiv.org/pdf/1312.4400.pdf.

[46] Adam Zewe. "Can machine-learning models overcome biased datasets?" In: (2022). URL: https://news.mit.edu/2022/machine-learning-biased-data-0221.

[47] Joseph Redmon et al. *You Only Look Once: Unified, Real-Time Object Detection*. 2016. arXiv: 1506.02640 [cs.CV].

[48] Anderson de Andrade. "This thing called Weight Decay". In: (2019). DOI: 10.48550/arXiv.1910.13029. URL: https://arxiv.org/ftp/arxiv/papers/1910/1910.13029.pdf.

[49] Griffin Hurt. "Normalization and Generalization in Deep Learning". In: *Rochester Institute of Technology* (2023). URL: https://repository.rit.edu/cgi/viewcontent.cgi?article=12521&context=theses.

[50] Yash Agrawal. "Batch Normalization in Neural Networks". In: 2018.

[51] Samuel L. Smith et al. "Don't Decay the Learning Rate, Increase the Batch Size". In: (2022). DOI: 10.48550/arXiv.1910.13029. URL: https://openreview.net/forum?id=B1Yy1BxCZ.

APPENDICES

A BasicModelMediumParams Architecture

```
class IncreasingModelMoreParams(nn.Module):
    def __init__(self):
        super().__init__()
        self.model = nn.Sequential(
            # input ch1,300x50
            nn.Conv2d(
                in_channels=1,
                out_channels=128,
                kernel_size=(3,3),
                stride=1,
                padding=(0,0)
                ),
            nn.ReLU(),

            # input ch64,298x48 -> 296x48
            nn.Conv2d(
                in_channels=128,
                out_channels=256,
                kernel_size=(3,3),
                stride=1,
                padding=(0,1)
                ),
            nn.ReLU(),
            nn.MaxPool2d(           # 296x48
                kernel_size=(2,2),
                padding=0,
                stride=2
                ),

            # input ch64,148x24
            nn.Conv2d(
                in_channels=256,
                out_channels=512,
                kernel_size=(5,5),
                stride=1,
                padding=(2,2)
```

```
            ),
        nn.ReLU(),

        # input ch64,148x24
        nn.Conv2d(
            in_channels=512,
            out_channels=64,
            kernel_size=(1,1),
            stride=1,
            padding=(0,0)
            ),
        nn.ReLU(),
        nn.MaxPool2d(
            kernel_size=(2,2),
            padding=0,
            stride=2
            ),

        # input ch128,74x12
        nn.Conv2d(
            in_channels=64,
            out_channels=256,
            kernel_size=(7,7),
            stride=1,
            padding=(3,3)
            ),
        nn.ReLU(),
        nn.MaxPool2d(
            kernel_size=(2,2),
            padding=0,
            stride=2
            ),

        # input ch256,37x6
        nn.Flatten(),
        nn.Linear(256 * (37*6), 12)
    )

def forward(self, x):
    return self.model(x)
```

B Training and Validation Loops

```
training_start_time = time.time()

train_loss_vals, train_acc_vals = [], []
val_loss_vals, val_acc_vals = [], []
acc_loss_train_results_list, acc_loss_val_results_list = [], []

if __name__ == "__main__":
    for epoch in range(num_epochs):
        model.train()
        total_train = 0
        correct_train = 0
        running_loss_train = 0.0
        for batch in train_loader:
            inputs, labels = batch
            inputs, labels = inputs.to(device), labels.to(device)

            train_preds = model(inputs)
            loss = loss_fn(train_preds, labels)

            optimizer.zero_grad()
            loss.backward()
            optimizer.step()

            _, predicted_train = torch.max(train_preds.data, 1)
            total_train += labels.size(0)
            correct_train += (predicted_train == labels).sum().item()
            running_loss_train += loss.item()

        train_acc = correct_train / total_train
        running_loss_train /= len(train_loader)

        train_loss_vals.append(running_loss_train)
        train_acc_vals.append(train_acc)

        print('Epoch [{}/{}], '
              'Training Loss: {:.4f}, '
              'Training Accuracy: {:.2f}%'.format(
                  epoch + 1,
                  num_epochs,
                  running_loss_train,
                  (correct_train / total_train) * 100
                  )
              )


        train_results_dict = {
```

```
            'model_name': model.__class__.__name__,
            'epoch_nb': epoch + 1,
            'train_accuracy': train_acc,
            'train_loss': running_loss_train
            }
        acc_loss_train_results_list.append(train_results_dict)


        model.eval()
        running_loss_val = 0.0
        correct_val, total_val = 0, 0
        with torch.no_grad():
            for batch in val_loader:
                inputs, labels = batch
                inputs, labels = inputs.to(device), labels.to(device)

                outputs = model(inputs)

                loss = loss_fn(outputs, labels)
                running_loss_val += loss.item()

                _, predicted = torch.max(outputs.data, 1)
                total_val += labels.size(0)
                correct_val += (predicted == labels).sum().item()

        val_acc = correct_val / total_val
        running_loss_val /= len(val_loader)

        val_loss_vals.append(running_loss_val)
        val_acc_vals.append(val_acc)

        print(f'Epoch [{epoch + 1}/{num_epochs}], '
              f'Validation Loss: {running_loss_val:.4f}, '
              f'Validation Accuracy: {(val_acc) * 100:.2f}%\n')

        val_results_dict = {
            'model_name': model.__class__.__name__,
            'epoch_nb': epoch + 1,
            'validation_accuracy': val_acc,
            'validation_loss': running_loss_val
            }
        acc_loss_val_results_list.append(val_results_dict)

model_time = (time.time() - training_start_time) / 60
print('Training finished, took {:.2f} minutes'.format(model_time))
```

C Accuracy and Loss Curves for Different Models

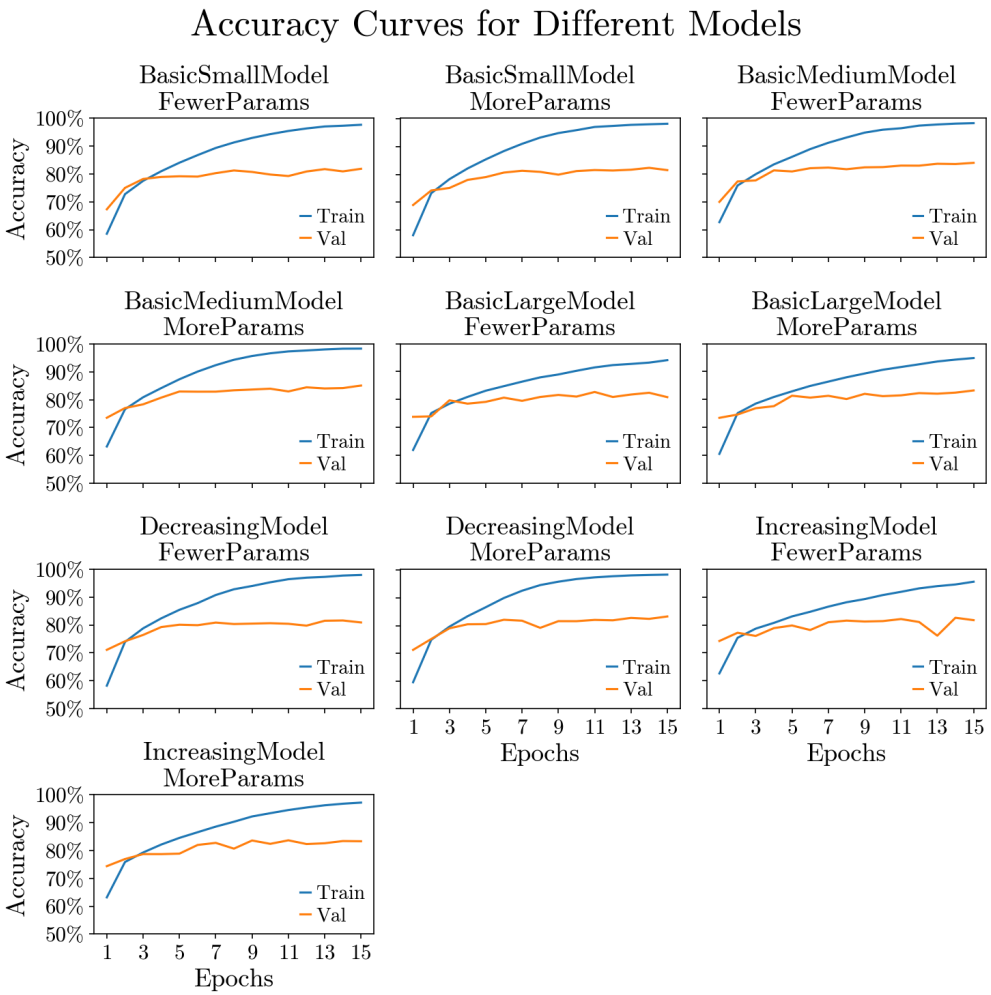

Figure 1: Accuracy curves for all models in Step 1.

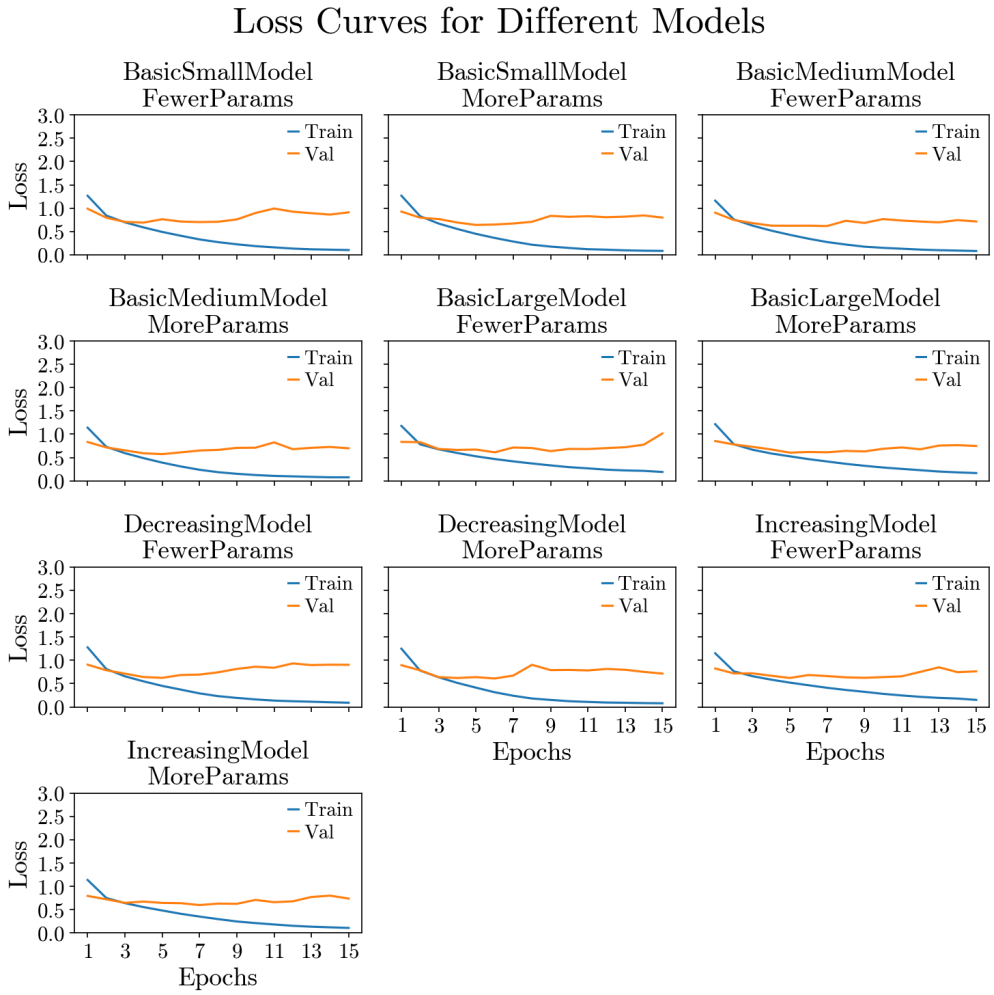

Figure 2: Loss curves for all models in Step 1.

D Learning Curves with Lower Learning Rate

The figures below display learning curves using the hyperparameters listed in the following table:

Table 1: Additional hyperparameters applied.

Hyperparameter	**Value**
Train set size	76.5 %
Validation set size	13.5 %
Test set size	10.0 %
Optimizer	RMSprop
Regularization	None
Model	BMMMP

The plots in Figure 3 look more like the "classic" learning curves one would often see. The configuration does not achieve a higher accuracy than the ones achieved in Chapter 4.

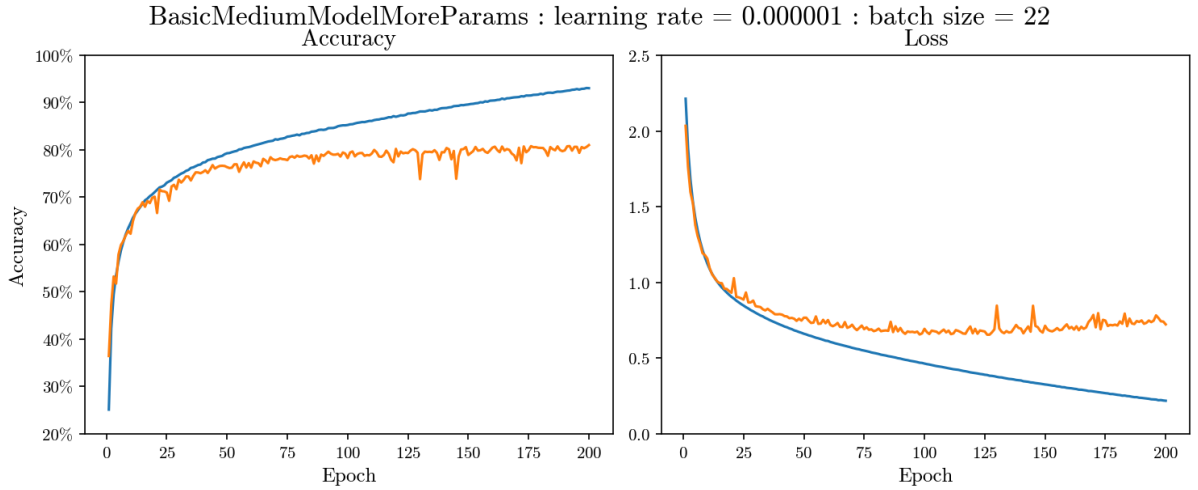

Figure 3: Learning curves. Left: batch size 32. Right: 128.

E Additional Incorrect Predictions