Tailoring the Hyperparameters of a Wide-Kernel Convolutional Neural Network to Fit Different Bearing Fault Vibration Datasets

Dan Hudson[a], Jurgen van den Hoogen[a], Martin Atzmueller[a,b]

[a]*Semantic Information Systems Group, Osnabrück University, Wachsbleiche 27, 49069, Osnabrück, Lower Saxony, Germany*
[b]*German Research Center for Artificial Intelligence (DFKI), Hamburger Straße 24 49084, Osnabrück, Lower Saxony, Germany*

Abstract

State-of-the-art algorithms are reported to be almost perfect at distinguishing the vibrations arising from healthy and damaged machine bearings, according to benchmark datasets at least. However, what about their application to new data? In this paper, we are able to confirm that neural networks for bearing fault detection can be crippled by incorrect hyperparameterisation, and also that the correct hyperparameter settings can actually change when transitioning to new data. The paper weaves together multiple methods to *explain* the behaviour of the hyperparameters of a wide-kernel convolutional neural network and how to set them. Since guidance already exists for generic hyperparameters like minibatch size, we focus on how to set architecture-specific hyperparameters such as the width of the convolutional kernels, a topic which might otherwise be obscure. We reflect different data properties by fusing information from seven different benchmark datasets, and our results show that the kernel size in the first layer in particular is sensitive to changes in the data. Looking deeper, we use manipulated copies of one dataset in an attempt to spot why the kernel size sometimes needs to change. The relevance of sampling rate is studied by using different levels of resampling, and spectral content is studied by increasingly filtering out high frequencies. At the end of our paper we conclude by stating clear guidance on how to set the hyperparameters of our neural network architecture.

Keywords: Deep Learning, CNN, Hyperparameter Analysis, Time Series Classification, Industrial Fault Detection, Bearing Fault Detection

1. Introduction

Machines are full of bearings, the components that make rotation possible. They are small metal balls usually arranged in a circular casing and by rolling, these spheres allow one piece of a machine to move around relative to another piece of the machine. Wear and tear unfortunately can accumulate over time into damage that causes the bearing element to ultimately break down, taking the entire machine offline with it. For example, one way this can happen is if small scratches develop in the casing that holds the bearings in place. As the ball bearings roll over that scratch again and again (perhaps hundreds of times per minute for days, weeks and months), it can grow into a structural weakness. One day, the casing can crack due to this weakness, breaking the rotating element and putting the entire machine out of commission until repairs are made. Therefore, methods for detecting such faults are crucial.

In this paper – a significantly extended and adapted revision of [1] building on our work presented in [2] – we tackle the question of how to configure neural networks so that they can effectively identify different types of damaged bearings. Thus, in this paper we do not start from scratch but instead our work takes place after and continues from research we published earlier. In particular, this paper presents a series of new experiments that address questions raised and left unresolved by our earlier work in [1] and [2]. Two new neural network architectures are studied in order to show how important hyperparameter configuration is to a wider range of networks. We present a novel experimentation methodology, extending the methodology presented in [2], for assessing the impact of hyperparameters in the context of wide-kernel approaches for bearing fault detection. Doing that, we run new experiments on manipulated subsets of the data to see how dataset properties affect the key question of how to configure hyperparameters. Specifically, we present more datasets, more neural network architectures and more extensive results compared to [2].

1.1. Finding Faults and Beating Benchmarks

From outside of the machine, the damage that develops in a bearing element cannot be seen. Bearings are encased within the machine and hidden from view. This creates obvious problems for monitoring the accumulation of damage within a bearing element and deciding when a replacement is needed. However, using vibrations which spread through the machine, we can try to look inside at what is happening: sensors placed on the outside of the machine can pick up vibrations that originate from deep inside. This is an approach that gives some hope of detecting when replacements are needed before a machine has an unexpected breakdown. Unfortunately, vibrations bring their own problems since reading these signals is not like looking at a photograph, and instead a lot of work is needed to interpret the vibrations that are being gathered by the sensors. This is in particular relevant to initiatives such as Industry 4.0 [3, 4], e. g., for enabling predictive maintenance applications [5], for which also explainable and interpretable approaches are usually beneficial [6].

One way to interpret vibration data and figure out how to distinguish healthy and damaged bearings is to learn from experience, by applying machine learning methods. Neural networks in particular are machine learning methods that can be very successfully trained by exposing them to a dataset of examples. As summarised in [7], various types of neural networks have been reported to achieve 97%+ accuracy on fault detection tasks, in papers such as [8, 9, 10, 11]. Furthermore, we also developed various deep learning algorithms to improve performance in fault detection, as described in [12, 13].

In the pursuit of effective algorithms to recognise when damage has occurred to a bearing element, researchers have settled on a series of 'benchmark' datasets, used to train and test different approaches. The obvious benefit of these openly-available datasets is that it is like having a fair race: everyone is starting at the same place and running the same route, so we can compare algorithms fairly based on their performance on these benchmark datasets. In our work, we make use of seven datasets originating from different organisations: Case Western Reserve University (CWRU), Paderborn University, the so-called 'Gearbox' dataset, the Society for Machine Failure Prevention Technology (MFPT), University of Conneticut (UoC), Southeast University (SEU) and the Xi'an Jiaotong University (XJTU).

Looking at past results on benchmark datasets, it has become obvious that multiple successful deep learning approaches exist. A recent review of the literature on the topic [7] indicates that at least 25 different deep learning algorithms have achieved over 95% accuracy (often 99% or 100%) on one or more of the benchmark datasets mentioned above. Such consistently high levels of performance imply that the difference between two state-of-the-art algorithms is now predestined to be small and there is little room left to improve beyond previous results. Therefore, comparing one state-of-the-art option to another is becoming less interesting and important than before.

Previous research in bearing fault detection has focused almost exclusively on finding algorithmic innovations with the hope of achieving higher scores during testing; however, since there is little room to improve, we now propose that the salient question is not whether new algorithms can be found, but whether the existing algorithms will continue to perform well when applied to new, unseen data. This is the question of whether they will still achieve near-perfect results when someone applies them in a real-life setting, when the sensors and machines are different from what the benchmarks used.

1.2. Limits of What the Benchmarks Can Tell Us

Benchmarks are very useful when comparing algorithms but they can only give an indication about how successful a method will be when applied to new data in the real world. The belief that an algorithm will perform well when it analyses a new set of machine vibrations, in a previously-unseen context, is simply an assumption and an assumption that can be challenged. For example, the types of damage that build up when a machine is really used in a factory might go beyond what is seen amongst the artificially-created damage recorded in benchmark datasets. Who can say that the types of damage found in the real factory will not be more subtle and hard to detect, or more diverse and thus requiring an algorithm with increased flexibility? Such questions start to introduce uncertainty. Differences in the properties of the recorded data might mean that an algorithmic approach does not generalise perfectly, leading to inferior performance when it is applied to a new dataset.

Speaking more generally about neural networks, when a practitioner takes an existing algorithm and tries using it on their own data, there is a risk that it does not work. Imagine that the practitioner is left with an algorithm that does not work. What can they do? One of the first things they can try is to modify the hyperparameters of the algorithm. In order to control the training of a neural network, the user must specify some so-called 'hyperparameters', such as the minibatch size (the number of examples used in a single step of training). These hyperparameters control how to generate the training signal that the neural network learns from – they are used in generating the numbers that incrementally update the neural network. Just like how a mis-tuned radio will not play your favourite radio show, it is possible to mis-tune hyperparameters and end up with a training signal that does not do what you want, i.e., it will not teach the neural network how to do the task.

1.3. The Open Question of Hyperparameters

Alongside hyperparameters that modulate the training process, one can consider what we might call 'architectural hyperparameters', which control the size and shape of a neural network. Quite commonly, when faced with poor results, a practitioner might make their neural network bigger. The layers used to construct the network could be enlarged or the network could be built by stacking more layers together. After making such changes, the neural network will contain more trainable parameters (numbers used to store information about how to process inputs) and thus have more capacity to learn about the task. If a neural network is not working then quite a natural step is to make it larger by increasing the architectural hyperparameters.

The reason we tweak hyperparameters is that the wrong choice of settings will lead to inferior performance whilst good choices will bring out the best from a neural network. In the case of severe ill-hyperparameterisation, a network might fail to learn at all during training. Returning specifically to bearing fault detection, we noted earlier that the differences appear to be minimal between the top 25 neural networks for detecting damaged bearings. By contrast, larger differences in accuracy can emerge when the hyperparameters of a network are changed. For example, [14] reports that the accuracy of a bearing fault classifying CNN is cut in half simply as a result of increasing from 6 convolutional layers to 8. When it comes to getting near-perfect accuracy on bearing fault detection, getting the hyperparameters right might be more important than choosing between neural network architectures.

However, although bad hyperparameter settings can result in misery, there is a question of why this matters, because we know that the hyperparameters work on our benchmark data. If the hyperparameters are already configured well, then what exactly is the problem? This is really a question of whether hyperparameters that work well on one dataset will necessarily be successful on another. To hone in on just a single reason why this might not always happen, we can ask whether drastically changing the sampling rate would impact what size the filters in a convolutional network should be.

Convolutional filters act like learned patterns that the network recognises at different time points during the input signal. Each pattern is stored in an array of numbers. If we increase the sampling rate used to record a signal that is otherwise kept fixed – so that more points are used to record the same peaks, troughs and plateaus over time – then it might be reasonable to expect that the number of points used to record the important patterns (in convolutional filters) also needs to be increased. The main point of this example is to highlight how data properties like the sampling rate might cause a meaningful difference in what hyperparameters are successful on different datasets. In fact, by examining the performance of neural networks on seven benchmark datasets, one important contribution of our work will be to show that hyperparameters working on one dataset sometimes stop working when changing to another dataset.

Another question is whether some hyperparameters can be changed without affecting the accuracy of the neural network on a specific dataset. Of course, it is possible to imagine that there is a miniscule 'sweet spot' of hyperparameters, a specific combination of numbers which is the only one that works. The reason this seems unlikely is that there is no obvious explanation for how the authors of successful algorithms were able to find this one-and-only sweet spot. Therefore, we instead expect that at least some of the hyperparameters can be modified without disastrous consequences for the accuracy of the neural network. Having more options in this way, it would be easier for an author to find a successful combination of hyperparameters without performing a rigorous search.

1.4. Shaping the Neural Network

Architectural hyperparameters like the number of layers in a network affect how complicated it is to process each input. Making the network smaller reduces the number of internal parameters it has, meaning that the computer needs to perform fewer arithmetic operations. This has a knock-on effect that impacts, for example, how long it takes to train the network. Smaller networks train faster and are quicker to generate predictions on new data. If some architectural hyperparameters can be reduced without harming the accuracy, it would be good to know this. Practitioners would benefit from knowing that they can make the network smaller and thus faster to train and use. Moreover, in real-world settings, there is the possibility of 'embedding' fault detection algorithms, so that they run on small devices situated in and amongst the machines they analyse rather than running on a user's laptop. Embedded processing skips the step of extracting data from the sensors and communicating it to a central location (like a laptop), and removes the need to keep a laptop or computer running continuously in order to process the data. An additional benefit of small networks is that they can be run on smaller and cheaper embedded devices. Clearly, there is real value to understanding the hyperparameters of a network, to see which hyperparameters can and cannot be changed when processing a given dataset. In this vein, we have worked to understand the architectural hyperparameters of a network capable of state-of-the-art results on fault detection tasks.

Of course, there must be limits to how much we can shrink a network or otherwise manipulate its hyperparameters. Some changes will be too drastic to work and they will make the neural network less able to detect bearing faults. Another line of our research is to study which hyperparameters are crucial and require the most careful handling. We wish to understand how much impact different changes will have and be able to state the relative importance of the hyperparameters. Re-examining the wide-kernel network presented in [15, 12], we distil new explanations of how the network's hyperparameters impact its accuracy at classifying bearing fault signals.

1.5. Do Dataset Properties Determine Performance?

Datasets have different properties, one reason why hyperparameters might need (re)tuning on new data even though they were already working on benchmarks. In the field of bearing vibration, research focuses on sensor recordings stored in 'time series', so called because they consist of a series of sequential measurements of some variable over time. The sampling rate is one of the key data properties when discussing time series. With a sampling rate of 12kHz, the movement of a vibrating machine is measured every 0.00008333 seconds. The sampling rate determines how many measurements are made per second and conversely how far apart in time each individual measurement point is. The sampling rate is documented for most benchmark datasets and we can see that it varies. One thread of our research will be to find out if changes to the sampling rate have an impact on the accuracy of a network and on which hyperparameter settings are best.

The sampling rate affects the dataset in multiple ways. It affects what frequency bands can be detected, with half the sampling rate (also known as the 'Nyquist frequency') providing the upper limit of what frequencies can be captured. An increased sampling rate also means that more numbers are used to record a vibrating machine, implying that the neural network needs to do more computation in order to process the inputs. One open question is how much the highest frequencies in vibration recordings are used by neural networks. This is interesting because if the highest frequencies are not needed, then storing and processing the recordings could be made more efficient by reducing the sampling rate. In fact, if high frequencies are unnecessary for fault detection then removing them could be a form of noise filtering. This is not implausible, since many signals resulting from vibrations (like from musical instruments) have an overtone series, implying that there is redundancy. As noted in [16], vibrations in bearing elements are transmitted to the sensors through the rest of the machine, which also consists of multiple components that themselves vibrate. This potentially leads to irrelevant or misleading information in the recorded signal, and if some frequency bands are just noise then it could be beneficial to remove them. Our research looks at how changes in the sampling frequency, for example when moving to one dataset to another one, affect neural networks. We ask whether filtering the irrelevant frequencies be helpful, for example by increasing the size of the 'sweet spot' of hyperparameters.

1.6. Explaining Hyperparameters

How many layers is enough in a neural network? Why do we set other hyperparameters in the way that we do? Although there are rules to follow when tuning some hyperparameters like the minibatch size (e.g., "set it to the largest value that will still allow the minibatch to fit into GPU memory") in other cases we can look at a neural network and find that the hyperparameter settings seem totally arbitrary. To pick just a pair of fault detection papers at random, we find that [9] uses 20 hidden units in fully-connected layers whilst [17] uses 64 instead. As is typical in the field, neither author noted why they chose the particular numbers they use and the reader is left with no explanation of why there is a difference. If there is ever a need to adapt these hyperparameters when working with new datasets, future users could quite justifiably feel lost. What users in fact want is a simple, explainable process for setting the hyperparameters.

The terms "Explainability" and "Explainable AI (XAI)" tie together several strands of research aimed at clarifying how the inputs to an algorithm relate to the outputs. With complex statistical models such as neural networks the inner workings are not understandable and we must rely upon specialised techniques to illuminate why particular outputs were chosen by the algorithm. This is a large topic and there are many surveys of the work done in this research area, such as [18] for example. What XAI aims to do is to cast a light on obscure and hidden relationships that exist in the algorithm and which link the inputs to the outputs. In many real-world domains such as bearing fault detection, algorithms need to learn nonlinear mathematical functions in order to approximate the complex interactions between multiple factors seen in observed data.

Returning now to our own focus, we want to do something similar with hyperparameters in order to better understand how they propel some networks to near-perfect accuracy and other networks to near-total failure. As we have noted, neural networks are clearly powerful and numerous in bearing fault classification but they are also poorly understood. By adapting techniques from XAI, we aim to build a bridge between empirical results and more general insights about how to tune hyperparameters.

1.7. The Paper Ahead

The present paper re-examines and then extends upon the results from several pieces of our own past work. In addition to providing new insights on past results, this present study presents new experiments with resampling and filtering of the data, allowing us to investigate how the properties of the data affect hyperparameter tuning.

To summarise the intention of the research: we already have neural network architectures for bearing fault detection ([7] gives many examples) but their success relies on hyperparameters which are poorly understood. Hyperparameters become especially important when we remember that the users of a neural network will not want to take it and apply it to a benchmark dataset. For the network to be any use at all, it must be possible for someone who is not the developer to take it and apply it themselves to novel datasets. To do so, they need some idea of how to decide the hyperparameter settings. However, the hyperparameters might as well be written in a cypher – there is no intuitive way to know that 64 convolutional filters works but a rate of 128 does not. The numbers seem arbitrary.

Engineers in a factory might have different machinery and types of bearings from what was used in benchmark datasets. They may have restrictions about where the sensors can be placed, and even what sensors they can use to record vibrations. Together, the specifics of a factory's machinery, bearing types, sensor availability and sensor placement will conspire to make a unique dataset for each engineer. As already noted, the choice of algorithm will not impact performance that much on benchmark datasets (see [7]), but the performance drop as a result of trying out a new dataset might be noticeable. Therefore, we take an existing neural network architecture which is capable of achieving state-of-the-art results on multiple benchmarks, and enrich it by presenting a detailed analysis of the hyperparameter decisions presented to the user.

We use a cocktail of multiple methods to explain the impact of hyperparameters on the wide-kernel CNN's accuracy. We summarise results from past work in order to see the average performance level when specific hyperparameter values are carried across seven datasets. We also harvest new results from experiments on manipulated versions of the data. We quantify the 'feature importance' – the extent to which different variables contribute to an outcome – of different hyperparameters and how it changes when processing either resampled or filtered copies of a dataset. This lets us find out how relevant dataset properties like the sampling rate are to deciding what hyperparameters to use. We also undertake separate additional experiments with two other architectures (besides the wide-kernel CNN) in order to round out the generality of our basic premise.

The key contributions of this work are:

- We show that hyperparameters are important across multiple types of neural networks rather than a single network type. Past results and new experiments with additional neural network architectures confirm this.
- We show that hyperparameters affect how accurate a neural network is (the majority of our work specifically investigates a wide-kernel CNN architecture).
- Fusing information from 7 datasets to analyse the wide-kernel architecture, we show that hyperparameters that are successful on one dataset are not necessarily successful on another dataset.
- We also highlight that not all hyperparameters are equally important.
- We propose that properties of the dataset, such as sampling rate, impact the optimal values of hyperparameters, which leads us to run experiments with manipulated versions of the time series data.
- After fusing the information from multiple different experiments, we find that due to redundancy in the signal, very high frequencies are not needed for successful bearing fault detection.
- We give general guidelines for how to set the hyperparameters of a wide-kernel CNN when transitioning to new data.

2. Background & Related Work

In this section, we review fault detection for the rotating machines seen in industrial settings. We also provide an overview of deep learning (DL) in the realm of time series analysis using sensor data, emphasising the design and training of convolutional neural networks (CNN). Additionally, given that this research concentrates on the architectural hyperparameters of a wide-kernel network, we discuss studies related to hyperparameter optimisation.

2.1. Fault Detection

Detecting faults in rotating industrial machinery is essential to avoid breakdowns [19]. Fault detection data are typically collected from sensors that measure vibrations, represented as time series data. Initially, fault detection relied on physics-based models that required an understanding of the underlying mechanisms that generate and propagate vibrations [20]. However, these models struggled to adapt to changing environments and increasing data complexity. The advent of the Industrial Internet of Things (IIoT) and data-driven analysis techniques has revolutionised fault detection methods, enabling a more intelligent and automated approach [17, 21]. These advances eliminate the need for in-depth technical knowledge of industrial machinery, allowing for automated processing and adaptation to changing operational environments. For example, machine learning approaches such as K-NN [22, 23], Random Forest [24], and SVM [25, 26, 27, 28] have been applied for fault detection. However, these methods require extensive feature extraction, a time-consuming process that has to be fine-tuned towards the type of data used.

With the development of deep learning techniques, these steps were no longer necessary since deep learning techniques are able to automatically extract features from raw data, simplifying the otherwise complex feature extraction process. Initially, multilayer perceptrons (consisting of computational units arranged into simple layers where every unit in one layer connects to every unit in the next layer) were used, but their limited depth due to computational constraints shifted the focus to other architectures such as recurrent neural networks (RNNs), which can more effectively model temporal dependencies [29, 30]. An RNN processes sequences of data one step at a time, passing information along internally to keep track of what it has seen. The computational units in an RNN are arranged differently than for an MLP, since the RNN is formed from small blocks suited for processing sequences one step at a time whilst the layers of an MLP must process the entire input sequence in a single vast computation. However, RNN networks are less suited for time series data that appear as especially long sequences, which is common in sensor data sampled at a high frequency (e.g., mechanical vibrations or electrical currents). The main reason for this relates to the storage of long-term dependencies, also resulting in increased computational needs when the time series is of high resolution. On the other hand, convolutional neural networks (CNNs) including the one-dimensional version for processing time series gained popularity, demonstrating state-of-the-art performance in fault detection, especially with wide-kernel designs in the first convolutional layer [31, 32, 2, 33, 34]. Convolutions are an efficient alternative to RNNs because they also process sequences as a collection of smaller sub-pieces.

2.2. Deep Learning

Deep Learning draws inspiration from the human brain's ability to combine many simple units into a large system capable of learning difficult tasks [35]. Its strength lies in processing and learning complex data, making it particularly effective for tasks requiring automatic feature extraction and refinement. This capability is crucial when valuable insights can be revealed by recognising patterns and relationships in the data [36, 37].

The fundamental component of a neural network is the artificial neuron, or "node," which processes multiple data inputs to produce an output. Many nodes can be grouped together into a layer and the term "deep" in deep learning refers to the inclusion of many layers in neural networks. A simple representation of a neuron i is given by:

$$a_i = \sigma\left(\sum_j w_{ij} x_j + b_i\right) \tag{1}$$

Here, x_j are the inputs, w_{ij} are the weights, b_i is the bias, and σ is a nonlinear activation function. Each input is multiplied by a corresponding weight and the resulting values are added together along with an extra bias term. The formula operates somewhat like a recipe that combines different amounts of the various inputs (the x_j terms) and adds some final garnish (the bias b_i). The nonlinearity introduced by the activation function is needed in order to handle difficult tasks, for example when similar inputs need different outputs. With layers stacking and interconnecting in complex architectures, neural networks can develop into intricate patterns, which gives them the flexibility and complexity needed to handle difficult tasks like image and speech recognition [36, 37, 38, 39].

The development of the multilayer perceptron (MLP) marked the beginning of deep learning, featuring fully connected layers [35, 40]. However, computational limitations initially restricted the depth of these networks. More advanced architectures were developed to overcome the limitations MLPs had when processing time series. Recurrent neural networks (RNNs), particularly long-short-term memory (LSTM) networks, emerged as effective architectures for time series analysis due to their ability to capture temporal dependencies [29, 30, 39]. Despite their success, RNNs are memory-intensive and less suited for long sequence data due to increased training times. RNNs are therefore not appropriate for directly processing raw inputs in many cases, and the undesirable step of pre-processing becomes necessary once again.

The initial application of MLPs in time series analysis spanned various domains, including stock prediction [41], weather forecasting [42], and fault detection [43]. Subsequently, RNNs [29, 39] and Convolutional Neural Networks (CNNs) [32, 21, 44, 17, 45] demonstrated significant performance improvements in time series tasks. CNNs, particularly one-dimensional (1D) CNNs were designed to process raw time series data effectively, leveraging automated feature extraction [31, 46]. These networks are robust against noise in the data and can be trained with relatively small datasets [47]. For fault detection a multi-layer perceptron (MLP) where all layers are fully connected [43] was initially proposed. Afterwards, the field of fault detection began to utilise convolutional neural networks (CNN) [17, 21, 44, 32] due to their notable performance improvements. CNN methods, when combined with data transformations like spectrograms, have been employed multiple times [48, 49]. However, 1D CNNs are able to process the raw time series data directly (without the need for spectrograms) and integrate automated feature extraction, and are often used in fault detection. These 1D CNNs also tend to be resilient to noise in time series data and can be trained with a small sample size [47].

Subsequently, numerous refinements for fault detection using convolutions were introduced. These include, but are not limited to, wide-kernel CNNs [15, 33, 34] (which are used in this work), attention CNNs [50], periodic CNNs [51], and CNN autoencoders for unsupervised learning tasks [52]. Each of these networks has unique characteristics regarding complexity, computational requirements, and specific tasks. The general mechanics behind CNNs are further discussed in the next paragraph.

2.2.1. Convolutional Neural Networks

Convolutional Neural Networks (CNNs) are a specialised type of neural network that were originally created to handle two-dimensional (2D) data, such as images. Introduced by LeCun in 1989 [53, 38], CNNs utilise a feed-forward architecture that applies convolutions instead of general matrix multiplications, making them highly effective for data organised in a grid-like structure. In contrast to conventional MLPs, CNNs have become essential in areas such as Computer Vision, Image Recognition [38, 37], and are increasingly used in time series analysis.

The primary benefits of CNNs compared to other neural networks arise from their use of local receptive fields, weight-sharing, and sub-sampling [54]. These characteristics greatly decrease memory usage and computational complexity, thereby improving algorithmic efficiency. Convolutional layers apply filters to convolve input data and this basically allows the network to identify where patterns occur within a large signal. The activation function, commonly a Rectified Linear Unit (ReLU), introduces non-linearity, allowing the network to learn intricate patterns [55]. After convolution and activation, a pooling layer often shrinks the signal's size in order to decrease the number of parameters of the next layer which helps prevent overfitting and lowers computational demands.

Initially, CNNs were less common for time series data due to their 2D nature, requiring conversion of one-dimensional (1D) data into 2D matrices, adding computational overhead. This changed with the development of 1D CNNs, which can directly process raw time series data [56, 34, 15, 47, 33, 31]. These advancements have solidified CNNs as a powerful tool for time series analysis, capable of efficiently handling high-frequency sensor data and extracting valuable features with minimal preprocessing.

The convolution operation is mathematically expressed as:

$$y_i^{l+1}(j) = k_i^l * M^l(j) + b_i^l, \tag{2}$$

where b_i^l is the bias, k_i^l represents the filter weights, and $M^l(j)$ is the local region of the input in layer l. The weights of the filter act in some ways like a learned pattern, in the sense that the filter reacts most strongly (gives the largest numbers as outputs) when the input correlates with the weights of the filter in addition to being large. The filter applies to little regions of the overall signal with each region being one stride along from the previous one. In the notation above i is a number used to refer to different regions. The most common activation function is ReLU [57], which introduces nonlinearity to the network.

Next, a pooling layer is utilised that allows the CNN to downsample the output of the convolutional layer making it more resource efficient in various applications from image and speech recognition to time series analysis.

2.3. Hyperparameter Search in Deep Learning

Past literature furnishes us with several examples of hyperparameter searches: activities in which authors tweak and tune the hyperparameters of a neural network in order to find the best performance. Because many decisions when designing and training neural networks can be called 'hyper-parameters', previous studies have different degrees of overlap.

Working in the field of sentiment analysis, [58] used a grid search – which takes a list of valid values for each hyperparameter individually and then tests out every possible combination of these values – to optimise the accuracy of their LSTM-CNN network. Their motivation for using the grid search approach was to guarantee the absolute best performance, which more efficient search methods trade off in favour of increased processing speed. As will be described in Section 3, we also employ a grid search method, although our reason for doing so is that it supports a wider range of follow-on analyses rather than to find a single combination of hyperparameters that is best. Grid search is fairly common, for example being used by [59] to optimise the three training hyperparameters of learning rate, minibatch size and dropout ratio when training a CNN. The authors of [59] used the Talos package to implement their hyperparameter grid search.

Bayesian optimisation is another popular technique to find good hyperparameter settings. When developing a CNN-GRU network to detect different types of activities from wearable fitness tracker-type devices, [60] used the Optuna and OptKeras packages to implement their Bayesian optimisation search. The hyperparameters they tuned included the number of convolutional filters in different layers, and the number of hidden units in the GRU layers. The authors of [61] used SigOpt to perform Bayesian optimisation on a wide range of hyperparameters, from minibatch size and learning rate to the number of fully-connected neurons and the properties of the convolutional layers in their network. Furthermore, [62] applied Bayesian optimisation for LSTM-RNNs in highway traffic prediction.

Another popular possibility for performing a hyperparameter search is to use a genetic algorithm. At the beginning of a genetic algorithm, there is a population of candidate solutions which are tested out. The key step is that, after testing out a population of candidates, a new 'generation' of candidate solutions is created to replace the old population of candidates, and the new generation is formed by combining and mutating candidates from the previous generation with some preference being given to the candidates that performed best. Such an algorithm is used by [63] to tune hyperparameters such as the number of layers, the number and size of convolutions in a layer, and the number of fully-connected neurons in a CNN for identifying sleep apnea. A genetic algorithm was used by [64] to decide a very similar collection of hyperparameters, although in this case the CNN was applied to emotion recognition in smart tutoring.

Beyond the search methods already mentioned, there are more esoteric ideas that are sometimes presented as alternatives. An example of a more uncommon approach is to use orthogonal arrays to decide which hyperparameter combinations to test out. In [65], experiments with RNN and CNN networks on three time series datasets found that a hyperparameter search based on orthogonal arrays got closer to the global maximum level of accuracy for a limited time budget, compared to random search and Bayesian optimisation alternatives.

Although these studies provided valuable insights, their methods and objectives differ from the present work, which seeks a comprehensive view of network performance through a grid search rather than focusing on optimisation algorithms. Hence, our approach emphasises an extensive automated grid search, inspired by the work in [2, 1] to determine if the most important hyperparameters change as a result of differences in times series properties. In this way, we attempt to better understand and *explain* the hyperparameters wide-kernel CNN.

2.4. Summary of Own Previous Work

In [2] we investigated how choices of the hyperparameters for a specific deep learning architecture (wide-kernel CNN) affect its performance in classifying faults from vibration signals in industrial machines. This architecture consists of five convolutional layers including an initial 'wide' layer with a large kernel size which then feed into a fully-connected classifier. See Section

3.3.1 for a detailed description. We were particularly interested in learning more about how performance changes across the hyperparameter space and so we performed a large-scale analysis, training and testing many versions of the wide-kernel CNN with different configurations (e.g., number of filters, kernel size) on three different datasets. The research used a grid search to explore the hyperparameter space and then presented multiple pieces of follow-on analysis.

Looking at how the distribution of test accuracy scores changes as a result of tweaking individual hyperparameters, it was found that the number of filters in the later convolutional layers (3 to 5) has the biggest impact on performance, with an optimal value around 32. The kernel size in the first layer was also found to be important for datasets with a high sampling rate, where larger kernels perform better. Another discovery was that high-performing hyperparameter settings could be unstable, i.e., small changes to hyperparameters can have a large impact on performance, highlighting the need for careful tuning.

Overall, this study provided insights into how to configure a wide-kernel CNN for better fault detection in industrial machinery. We also identified interesting areas for future research, such as applying this approach to other signal classification tasks and different neural network architectures.

A second study increased the number of benchmark datasets by four and applied 12,960 variations of the wide-kernel CNN to seven different industrial vibration datasets [1]. Once again, we analysed the grid search in order to identify how hyperparameter settings influence the network's performance and how to effectively tune them for new datasets.

Compared to the previous study [1], a more thorough investigation using seven datasets confirmed that the most important hyperparameters for the network's performance are the number of filters in layers 3-5 and the kernel size in the first layer. New analysis focused on pairwise interactions between hyperparameters, where changing one hyperparameter might require adjusting another one for optimal performance. After looking at how hyperparameters influence one another, we suggested a specific order for tuning the hyperparameters to minimise the need to re-tune them later. This would be useful in situations where practitioners want to tune the hyperparameters one-at-a-time sequentially rather than using a full grid search. Tuning the hyperparameters in our suggested order leads to better performance compared to random or reverse order.

3. Method

Our research method consists of several parts which feed into each other. In this section, we step through each piece of the method and describe it in detail. Starting out, we describe seven different benchmark datasets that we use to train and test neural networks. Different types of fault, machinery and recording set-up are possible in fault detection and we therefore expect that datasets can differ from one another. We want to account for variations in the data and so we assembled a selection of seven benchmarks to train and test out CNNs. The properties of the benchmark datasets are described in detail below.

Next, our method splits into two main structural components based on which architectures we study. Previous work in [2] and [1] looked at the consequences of tuning the hyperparameters of a wide-kernel CNN but not other kinds of architecture. We devote a short section to the impact of hyperparameter tuning on two other architectures with the goal of showing more generally that hyperparameters matter for bearing fault classification, at least when using neural networks. We introduce two popular types of architecture that are used in this short section as alternatives to a wide-kernel CNN. Additionally, we describe how we generate results by sampling different hyperparameter values from a range of possibilities and then training and testing networks on the seven benchmark datasets.

The second structural component and the majority of the research however focuses on the wide-kernel CNN and aims to investigate how much dataset properties affect which hyperparameter settings are best. We describe the wide-kernel CNN architecture used in our work and its architectural hyperparameters. Training and testing on seven benchmark datasets gives an indication of how differences in the data can have consequences for hyperparameter tuning. We start off with a grid search which is an exhaustive exploration of different combinations of hyperparameters and for each combination we train and test a CNN on benchmark data. We repeat the grid search seven times in order to include all the benchmark datasets mentioned earlier.

In addition to accepting the benchmark datasets as-is, we also want to get into a position to see what happens when we manipulate the dataset properties ourselves. By manipulating the datasets, we gain fine-grained control over factors that might explain why some hyperparameter settings result in high accuracy while others perform badly. In concrete terms, we edit the sampling frequency of the CWRU dataset by resampling it and we also cut out high frequencies by filtering. Both types of data manipulation are described below. Each modified version of the CWRU data becomes the input to a new grid search over important hyperparameters and produces a new grid of test accuracy scores. Collating these grid searches, we can compare how hyperparameters behave in response to different levels of resampling or filtering. At this point, we have several different types of architecture being tested out on different datasets; for clarity, we illustrate the overall flow of the various steps in Figure 1.

Ultimately we wish to extract useful insights into how to tune hyperparameters. There are actually several related analyses based on the outputs of the grid searches, and we go over each one in turn. Correlations are used to see how much hyperparameter settings generalise across datasets. Feature importance is used to see which hyperparameters matter the most. We have two approaches, the first of which is based on Shapley values and the second of which is based on the d-dimensional earth mover's score. What these terms mean is laid out towards the end of this section.

3.1. Datasets

In this subsection we describe the most obvious input to our experiments: data. We use seven benchmark datasets, described below along with their main properties of the datasets and our pre-processing steps. These are the seven datasets we used earlier in [1].

3.1.1. CWRU Bearing Dataset

The CWRU bearing dataset, provided by Case Western Reserve University [66], represents a benchmark for fault detection experiments where various damages were gouged into bearing elements. Sensors were placed on the drive end and fan end of the machine to measure vibration signals, which were digitised into two time series and segmented into sequences of 2048 data points. The data was collected at a sampling rate of 12 kHz for both the drive-end and fan-end experiment, and the data sampled at 48 kHz for the drive-end experiment. The latter is sampled at a much higher rate and therefore contains more data points. For all recordings retrieved from the CWRU experiments, we extracted only the machine operation conditions with motor speed of 1797 and 1750 rotations per minute (RPM). The damages of the bearing are inflicted at three depths (0.007, 0.014, and 0.021 inches) and across five fault locations (ball, inner race, outer race opposite, outer race

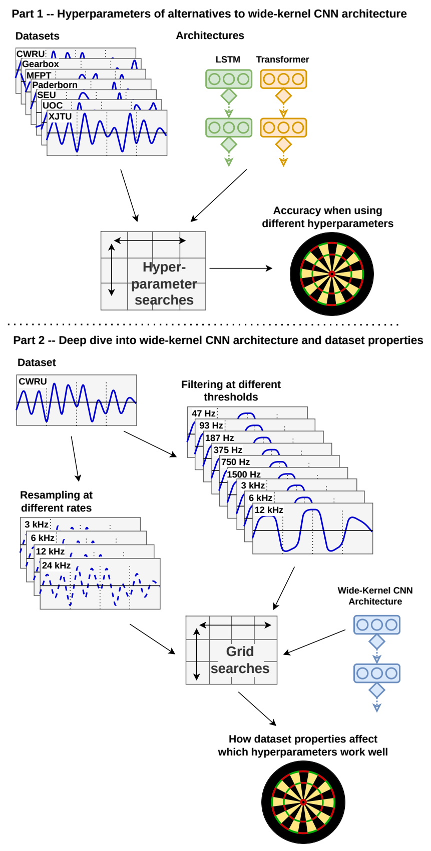
Part 1 -- Hyperparameters of alternatives to wide-kernel CNN architecture
Datasets
Architectures
CWRU
Gearbox
MFPT
Paderborn
SEU
UOC
XJTU
LSTM
Transformer
Hyper-parameter searches
Accuracy when using different hyperparameters
Part 2 -- Deep dive into wide-kernel CNN architecture and dataset properties
Dataset
CWRU
Filtering at different thresholds
47 Hz
93 Hz
187 Hz
375 Hz
750 Hz
1500 Hz
3 kHz
6 kHz
12 kHz
Resampling at different rates
3 kHz
6 kHz
12 kHz
24 kHz
Wide-Kernel CNN Architecture
Grid searches
How dataset properties affect which hyperparameters work well

Figure 1: Workflow of the method to test out multiple architectures on varying data.

orthogonal, and outer race centered) with approximately balanced samples per class except for the normal conditions. We concatenated fault conditions that were equal between the two motor speeds since they are in fact the same fault. However, in some cases, a fault did not occur in both machine operating conditions, therefore making the data somewhat imbalanced. In total for all three experiments, we identified 13 fault conditions.

The specific choice for this dataset relates to the frequent usage within the fault detection domain [15, 47, 49, 67, 68, 69], due to its public availability and its structure, which mirrors real-world industrial applications where data is typically not openly accessible. For training purposes, we split the dataset into just 20% train and 80% test data in order to enhance the complexity of the classification task and reduce computation time. For more detailed information, we refer to the original source [66] and the test rig displayed at `https://engineering.case.edu/bearingdatacenter/apparatus-and-procedures`, along with Figure 2 which shows an example of the vibration signals.

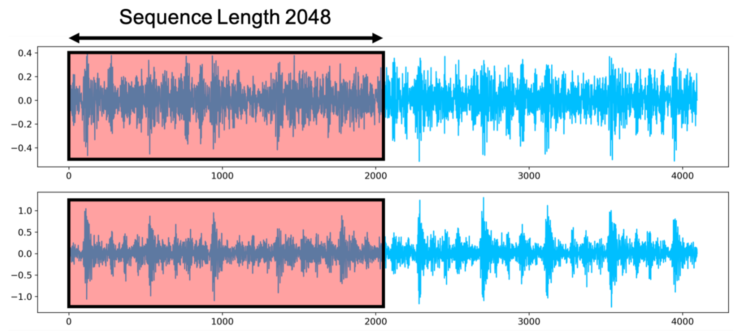

Figure 2: Illustration of vibration signals retrieved from [12], representing two sensors based on the CWRU Bearing dataset. The red box highlights the duration of a single data sample (sequence).

3.1.2. Paderborn Dataset

The Paderborn bearing dataset serves as a benchmark for fault detection and condition monitoring of damaged rolling bearing elements, as seen in studies like [17, 44]. The dataset captures motor current signals of an electromechanical drive system and vibrations of the housing [70]. These signals are extracted using existing frequency inverters, eliminating the need for additional sensors, which was necessary in the CWRU bearing experiments. This results in more resource-efficient and cost-effective experimentation. A unique feature of the method is its ability to monitor damages in external bearings positioned within the drive system but outside the electric motor.

In total, the data derived from the experiments represents “healthy”, “real damaged”, and “artificially damaged” bearings. The data is recorded for approximately 4 seconds at a sampling rate of 64 kHz which produces numerous data files containing around 256,000 data points each. For this study, we focused solely on the “real damaged” experiments, specifically targeting the “inner race” faults, encompassing 8 distinct conditions as highlighted by [70].

This resulted in 80,000 sequences (10,000 for each condition), necessitating substantial computational resources to run various CNN configurations. To manage this, we took an equally divided random sample from the dataset, retaining 10%, or 1,000 sequences per condition. Similar to other datasets, a train/test split of 20% and 80% was implemented.

The Paderborn Bearing dataset, compared to the CWRU data, has fewer classes, more balanced data between classes, and a higher number of sequences, which increases network training time. Furthermore, the dataset includes three time series, two motor current signals and one vibration signal. We only used the vibration signal to align with the data types from the other datasets. For this experiment, we used k-fold cross-validation with $k = 3$ to enhance the generalisability of the trained networks and tested the networks' performance with various data splits.

3.1.3. Gearbox

The Gearbox dataset originates from the necessity to optimise and analyse industrial gearboxes, particularly in applications like wind turbines where gearbox failures can lead to significant downtime [71]. Previous research has shown that downtime due to gearbox failures is higher compared to other components. The most common approach for examining faults in wind turbine gearboxes involves recording and analysing vibration signals, which are inherently non-stationary. The Gearbox dataset was created using the SpectraQuest Gearbox Fault Diagnostics Simulator, enabling researchers to simulate the behavior of an industrial gearbox for diagnostics and prognostics research [72]. The dataset includes vibration signals from four sensors placed in various directions around the gearbox. The experiments were conducted under varying load conditions, ranging from 0% to 90%. The dataset encompasses two primary conditions: a healthy gearbox and one with a broken tooth. We combined all load variations into a binary classification task, distinguishing between healthy and broken gearboxes. This resulted in 978 sequences, each containing 2048 data points. The dataset is balanced, with 492 sequences labeled as healthy and 486 as broken. For our study, we focused on these binary conditions and implemented a train/test split of 20/80% to make the task more challenging and to reduce computation times. Additionally, the dataset allows for the separation of each operating condition combined with their respective gearbox condition, offering a nuanced perspective on the data.

This comprehensive dataset provides a robust foundation for analysing and improving fault detection in industrial gearboxes, particularly in high-stakes applications like wind turbines. The detailed setup and balanced data ensure that the classification task, although simplified by the train/test split, remains a challenging and informative endeavor for network training and evaluation.

3.1.4. MFPT Dataset

The Society for Machinery Failure Prevention Technology (MFPT) dataset [73] is a widely used dataset for studying faults in machinery. This dataset comprises 7 outer race faults, 7 inner race faults, and a healthy baseline condition, resulting in a total of 15 unique classes. The data was collected under various load conditions to provide a comprehensive overview of fault scenarios.

Sequences of 2048 data points were created from one of the vibration sensors. Fault conditions were sampled at a frequency of 48.828 kHz, yielding 71 sequences for each fault type. In contrast, the healthy condition was sampled at a higher frequency of 97.656 kHz, resulting in an additional 858 sequences. This discrepancy in the number of sequences per condition makes the dataset inherently unbalanced.

The unbalanced nature of the dataset poses a challenge for machine learning algorithms, as it requires strategies to handle the imbalance effectively.

This dataset provides a rich source of information for the development and evaluation of fault detection algorithms, making it an essential tool for researchers in the field of machinery failure prevention. The diversity of fault conditions and the high sampling rates ensure that the dataset can be used to test a wide range of diagnostic methods.

3.1.5. XJTU

The XJTU-SY bearing datasets are provided by the Institute of Design Science and Basic Component at Xi'an Jiaotong University (XJTU) and the Changxing Sumyoung Technology Co. The datasets contain complete run-to-failure data of 15 rolling element bearings that were acquired by conducting many accelerated degradation experiments [74]. In this case, we chose to extract the last 30 minutes of every experiment containing the fault occurrence, and potentially already some vibration deviations prior to this fault. Therefore, sampling an equal length of data, containing the fault would be an appropriate strategy. Since the data are retrieved from a remaining useful life (RUL) experiment, the datasets are imbalanced in general, and also until the fault occurs, they are somewhat similar. The experiments contain two time series making it a multivariate use-case. The data are sampled at a 25.6 kHz where the sampling frame was 1.28 seconds within every minute, resulting 32,768 data points per minute. This results in 480 sequences for every of the 15 different fault conditions, totaling the dataset with 7,200 sequences. Due to the vast amount of data, we can state that this dataset is large.

3.1.6. UoC

The dataset retrieved from the University of Conneticut (UoC) [75] provides a gear fault dataset that measures vibrations with the use of accelerometers. the data are collected with a sampling frequency of 20 kHz. It only consists of a single sensor, therefore making it a single time series channel, i.e., univariate time series. In total, the dataset contains eight different gear fault conditions accompanied by one healthy gear condition, resulting in a multi-class classification task with 9 unique conditions. The following conditions are gathered during the experiments; healthy condition, missing tooth, root crack, spalling, and chipping tip with 5 different levels of severity [76]. The dataset is balanced for all conditions. We segmented the data into sequences of 2048 data points per sequence, giving 182 sequences per condition, amounting to a fairly small dataset.

3.1.7. SEU

The Southeast University (SEU) in China has developed two sub-datasets for their gearbox datasets, both aimed at providing insights into the health of bearings and gearboxes [77]. Data collection was accomplished through a Drivetrain Dynamics Simulator, capturing eight channels of vibration data. The data encompasses five different conditions: one healthy state and four fault states, all under two operational conditions defined by rotational speed and load: 20 Hz-0 V and 30 Hz-2 V. We specifically utilised the bearing-related data, extracting three vibration channels (channels 2, 3, and 4) corresponding to the x, y, and z directions of the planetary gearbox, thus forming multivariate time series data. To increase the complexity within every fault condition, we merged the data from both rotational speeds and loads. The full recordings were employed, including the initial start-up phase. Overall, the dataset consists of 5110 sequences of 2048 data points each, which translates to 1022 sequences per class, ensuring a balanced dataset across all conditions.

3.2. Method Pt. 1 – Experiments on Alternatives to Wide-Kernel CNN Architecture

A wide-kernel CNN architecture takes centre stage in our research however we also want to say that hyperparameters are an important and unavoidable decision for other neural networks too. Therefore, in addition to the wide-kernel CNN which is our focus we also perform limited experiments on two additional architectures. By encompassing more than one architectural style, our work has a broader basis on which more general conclusions can be built.

The first additional architecture uses “long- short-term memory” (LSTM) to digest the signal as a sequence of smaller pieces. The second additional architecture, a “transformer”, also consumes its inputs as a sequence. Both architectures boast a good record of accomplishing sequence processing tasks and can be adapted to work on time series.

Our goals for the LSTM and transformer architectures are not the same as those we have for the wide-kernel CNN. With the wide-kernel CNN, our intention is to weigh up the importance of different hyperparameters and familiarise ourselves with each hyperparameter in detail. Our results do not generalise to other architectures which have their own distinct sets of hyperparameters. The extra LSTM and transformer architectures are not included to allow for a detailed analysis but simply to provide evidence that hyperparameter tuning has important consequences for a wide range of state-of-the-art neural networks. We want to show that the dangers of ill-hyperparameterisation are not a consequence of choosing the wide-kernel architecture in particular.

3.2.1. Long Short Term Memory (LSTM) Architecture

Following the example of [78], we consider a bearing fault classifier built around Long Short Term Memory (LSTM) layers. The central idea of an LSTM layer is that it passes information forward between steps when processing a sequence stepwise – making it possible to ‘remember’ things seen earlier – and yet it also allows the internal flow of information to be flushed out [79]. Strategically flushing out the internal information deals with the problems of vanishing and exploding gradients that previous kinds of recurrent network suffered from, with the upshot being that LSTMs are more successful at handling longer sequences and performing a wide range of tasks.

Like [78], we situate LSTM layers in an architecture that can be imagined as a structure with three main chambers. The first two components are CNNs which proceed in parallel and then they both pass into the third main component which is the LSTM itself. Two CNNs are used in order to achieve a “multi-scale” effect. A CNN containing smaller kernels gives prominence to brief or high-frequency vibrations in the signal. Features of the signal that develop more slowly are emphasised in a second CNN consisting of larger kernels. The patterns that emerge from both CNNs then proceed into the

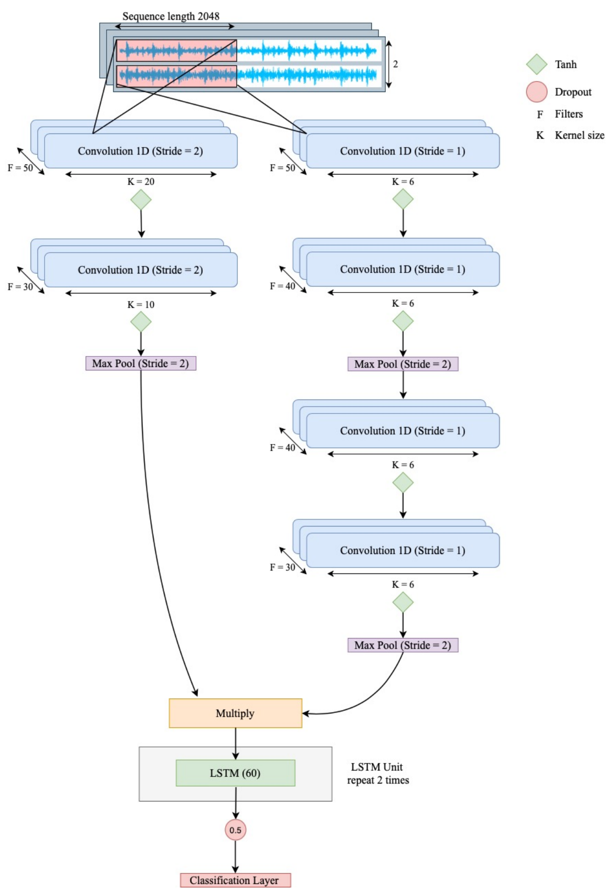

Figure 3: Architecture of the used LSTM network, inspired by [78]. The network utilises two convolutional paths at different scales which are multiplied together and fed into the main LSTM.

LSTM. Crucially, the patterns coming from the CNNs take on the form of a sequence, in which each step summarises a period of time from the raw signal. This is needed because the LSTM layer would be overwhelmed by the sheer number of measurement points in the raw signal without any summarisation. The LSTM component takes the shape of a pair of layers which run along the length of the input sequence and ultimately exit to a vector of numbers which captures the key details of the signal. At the end of the LSTM, fully connected layers head onwards to arrive at a final prediction of what type of fault is present in the machinery. The architecture is visualised in Figure 3

3.2.2. Transformer Architecture

We also consider a transformer architecture, following the example set by [50] for bearing fault analysis. Transformers use a so-called ‘attention’ mechanism which allows them to process each step in a sequence by pooling information from other steps [80]. The innovation of using attention provides an efficient alternative to recurrent connections, and the booming popularity of transformers in applications like ChatGPT testifies to their efficacy.

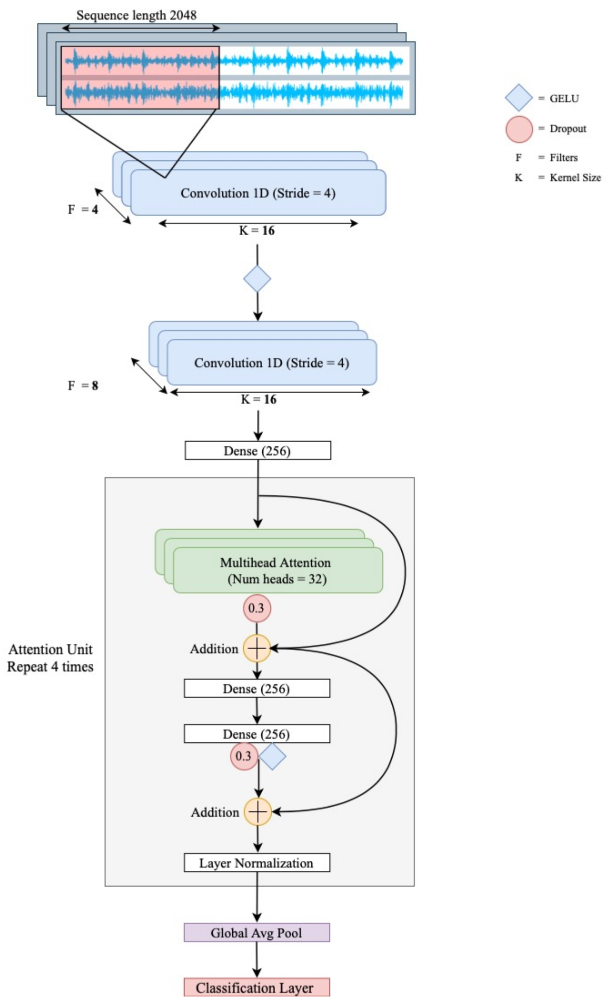

Figure 4: Architecture of the used transformer network, inspired by [50]. The network utilises convolutional layers that feed into transformer layers.

Much like the LSTM architecture, the transformer would also be quickly overwhelmed by the number of time steps in the raw vibration signal. Therefore, the network we use (shown in Figure 4) receives incoming signals in a CNN component which leads to a compact sequence that then passes through to the transformer component. The CNN has a simple structure arranged into two layers and after passing though it the raw input signal shrinks to a sequence of roughly 1/16th the length. The transformer section of the architecture is larger and consists of 4 layers by default. When exiting the transformer layers we still have a sequence of vectors. All the pieces of the sequence need to come together into a single vector, something that is achieved using a global average pooling layer. Following the pooling layer it is possible to proceed into a single fully-connected layer which brings us to the final prediction of the network.

3.2.3. Hyperparameter Search on Other Architectures

When studying the LSTM and transformer networks we restrict ourselves to drawing some generic conclusions about the impact of (mis)tuning hyperparameters. As such, we do not find it necessary to perform a full grid search, but instead to rely on a reduced sample from a grid of possibilities. We select 100 configurations uniformly at random from a larger collection of possible hyperparameter combinations.

For the LSTM-based neural network, the values we consider for each hyperparameter are shown in Table 1. The middle value for each hyperparameter is based on what it is set to by default in the code published alongside [78]. The other two options simply expand or shrink the amount by a factor of four. From the space of possibilities defined in Table 1, we take 100 random samples.

The transformer hyperparameters are also confined to a limited set of possibilities, which are shown in Table 2. From these we choose a sample of 100 combinations of values. Like with the LSTM, our choice of default options is inspired by the code published by the authors (alongside [50]).

3.2.4. Analysis Techniques

Our analysis of the LSTM and transformer architectures asks how much the accuracy of a network changes when its hyperparameters change. The evidence considered is a sample of 100 LSTMs and 100 transformers chosen at random from the range of hyperparameter options, with each network

Table 1: Hyperparameter values considered for the LSTM network.

Hyperparameter	**Value Domain**
Kernel size shallow network layer 1	{5, 20, 80}
Filters shallow network layer 1	{13, 50, 200}
Kernel size shallow network layer 2	Always half of layer 1 value
Filters shallow network layer 2	{8, 30, 120}
Kernel size deep network layer 1	{2, 6, 24}
Filters deep network layer 1	{13, 50, 200}
Kernel size deep network layer 2	Always equal to layer 1 value
Filters deep network layer 2	{10, 40, 160}
Kernel size deep network layer 3	Always equal to layer 1 value
Filters deep network layer 3	{8, 30, 120}
Kernel size deep network layer 4	Always equal to layer 1 value
Filters deep network layer 4	Always equal to filters in layer 2 of the shallow network
Hidden units LSTM layer 1	{15, 60, 240}
Hidden units LSTM layer 2	{15, 60, 240}

trained and evaluated on seven benchmark datasets. Based on their hyperparameters, some of the 200 networks perform better and some perform worse. To give an overview of the results, we present to the reader a summary of the distribution of accuracy scores. We report the distribution of test accuracy – the spread of how well the networks performed – as follows. First, for the LSTM we state the minimum, 25th percentile, median, 75th percentile and maximum accuracy achieved after training and testing on CWRU data. We then state what values the LSTM reached on the Gearbox data for the minimum, 25th percentile, median, 75th percentile and maximum (also known as Tukey's five-number summary). The results proceed to give the corresponding values for all the other benchmark datasets and when using the transformer instead. In total, there are 5 values describing how 2 architec-

Table 2: Hyperparameter values considered for the transformer network.

Hyperparameter	**Value Domain**
Kernel size CNN layer 1	{4, 16, 64}
Filters CNN layer 1	{1, 4, 16}
Kernel size CNN layer 2	{4, 16, 64}
Filters CNN layer 2	{2, 8, 32}
Hidden units in transformer layers	{64, 256, 1024}
Attention heads in transformer layers	{8, 32, 128}
Number of transformer layers	{1, 4, 16}

tures did on 7 different datasets, with the goal being to give an idea of the spread of the data and quickly clarifies how much the architectures are affected by their hyperparameters and by changes in the data. We are also curious how well the default hyperparameter values used in past literature fare. Therefore we also report the accuracy obtained when using the default settings.

3.3. Method Pt. 2 – Focusing on the Wide-Kernel CNN Architecture

Next we leave behind the LSTM and transformer architecture and arrive at the main body of the research, which is to explain how to set the hyperparameters of a wide-kernel CNN architecture.

3.3.1. Overview of Architecture

Previous work demonstrated excellent performance in bearing fault detection by employing a wide-kernel in the first convolutional layer followed by smaller kernels in subsequent layers [33, 15, 34, 2]. This wide-kernel CNN does not only perform well but is also particularly suitable for fault detection tasks due to its reasonable size in terms of convolutional layers and its capability to process raw time series directly.

Our architecture contains two convolutional layers, each utilising several combinations of stride, kernel size and filters. After that, a **convolutional unit** (see Figure 5) is repeated *three* times, resulting in a total network of five convolutional layers. The three convolutional units all have the same hyperparameter combination in this study, relying on the observation that the deeper layers of the network should have the same properties for optimal learning, as described in previous work [33, 34, 15]. After each convolutional layer, the network utilises local average pooling to decrease the vector size of the convolutional output with length T divided by two, resulting in a pooled output length of $\frac{T}{2}$. At last, the reduced output from the convolutional layers is fed to two fully connected layers and the last one functions as the classification layer. Figure 5 shows the overall network architecture with two exemplary time series as input.

We originally chose to use a wide-kernel CNN architecture as the focus of our hyperparameter investigation because it had already been achieved near-perfect accuracy on the two bearing fault detection datasets (CWRU and Paderborn) in prior work [15, 33, 34]. Its performance on these benchmarks indicated to us that it is well-suited for fault detection tasks and that a more complicated network would not necessarily give extra benefit. Furthermore, the architecture is lightweight in terms of computational resources and memory usage, which allows it to have a wider range of uses, in particular opening up the possibility of edge computing in the context of Industry 4.0.

Moreover, the modest depth of the wide-kernel architecture, compared to other state-of-the-art CNN architectures [51, 50, 81, 52, 78, 82], lends itself well for an extensive grid search of the hyperparameters due to the relatively low number of hyperparameters.

In addition to the network's architecture, some additional refinements improve performance and computation time. These remained the same for every combination that was tested. First of all, between convolutional layers, a batch normalisation layer is added to speed up the training process. Second, we used the Adam stochastic optimiser. Adam is an optimisation algorithm that leverages the strength of adaptive learning rates for each individual parameter and is well-suited for networks that analyse high dimensional input and have a lot of trainable parameters, since it is memory-light and computationally efficient. Furthermore, according to [83], Adam is very effective with noisy signals.

3.3.2. Hyperparameter Search

Each hyperparameter of a neural network is a decision and changing any one of them leads to a new configuration for the network, potentially leading to a different accuracy score on the task being attempted. From any given starting point there are as many directions for the configuration of the network to go as there are hyperparameters. By imposing lower and upper

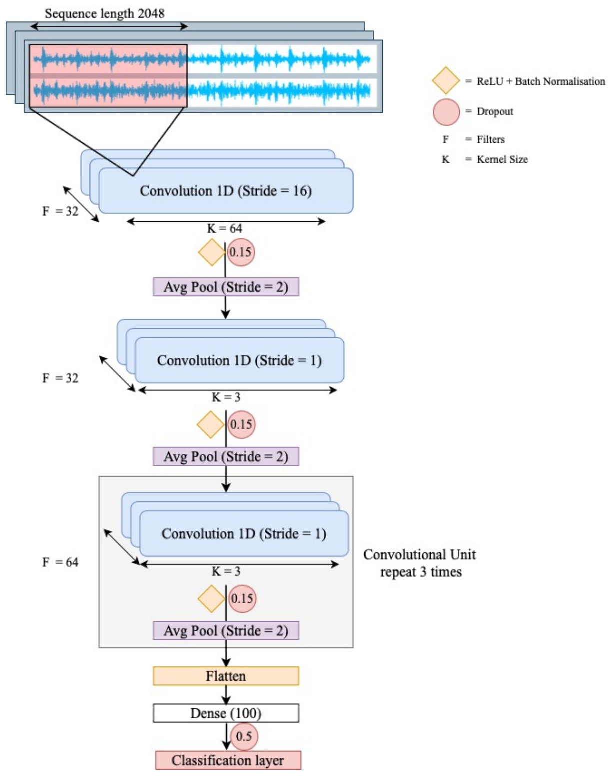

Figure 5: Architecture of the used 1D wide-kernel CNN, inspired by [33, 2]. The network utilises two convolutional layers followed by a convolutional unit (in grey), which is repeated 3 times.

limits on what values each hyperparameter can take, it is possible to chart out a finite hyperparameter space for exploration. In the ensuing section of this paper, we describe the hyperparameter spaces used in our different experiments and how we explore them.

Earlier work on the wide-kernel CNN identified seven architectural hyperparameters [2]. These were explored using a grid search strategy in which each hyperparameter is restricted to a limited set of values and every possible way of combining these is tested. For example, every kernel size in the first layer is tried out with every stride in layer 1, and each of these combinations is tried out with every kernel size in layer 2, etc. Since they differ from one another, the set of options tried for any particular hyperparameter is not the same as for other hyperparameters. Table 3 shows the options. Looking at all the mixtures that can be created, there are $5 \times 3 \times 6 \times 2 \times 6 \times 2 \times 6 = 12960$ combinations included in the grid search.

Previous work also found that three hyperparameters in particular are important: kernel size in the first layer, number of filters in the first layer and number of filters in layers 3-5 (these are indicated in italics in 3) [2, 1]. We use these hyperparameters as a basis for new grid searches. The reason we are performing new grid searches is that we want to find out more about the impact of manipulating the properties of the data by filtering or resampling. Our approach to filtering and resampling the data is described in the next subsections. By manipulating the dataset we create new versions of it. On each version, we perform a grid search considering the values shown in Table 3. The data manipulation strategies and the goals of performing them are explained in more detail in the next subsection.

In summary, we have a grid search performed over seven hyperparameters which tests the accuracy of the network on seven benchmark datasets. Additionally, as will be explained in a moment, we have multiple smaller grid searches which apply to modified versions of the data. These smaller searches operate over three hyperparameters.

Table 3: Architectural hyperparameters with the values they take in grid search.

Hyperparameter	**Value Domain**
Kernel size layer 1	{16, 32, 64, 128, 256}
Stride layer 1	{4, 8, 16}
Filters layer 1	{8, 16, 32, 64, 128, 256}
Kernel size layer 2	{3, 6}
Filters layer 2	{8, 16, 32, 64, 128, 256}
Kernel size layers 3-5	{3, 6}
Filters layers 3-5	{8, 16, 32, 64, 128, 256}

3.3.3. Analysis Techniques on Seven Benchmark Datasets

In order to flesh out certain points from the work reported earlier in [2, 1], we take the results from a grid search over seven hyperparameters applied to seven benchmark datasets and present them in new ways, as follows.

Descriptive Analysis:

If possible, we would like to recommend effective default values for the seven architectural hyperparameters of our wide-kernel CNN architecture. Recommendations would be easy to make if we find that a single value performs best across all the seven benchmark datasets. Therefore, we present a simple table that shows the average accuracy obtained on each dataset when using each hyperparameter value. The table compares different options for a hyperparameter, showing if there is a generally best option or if instead there is no obvious default value that performs well on all datasets.

Influence between Important Hyperparameters:

Part of the work of [1] was to attribute importance scores to the hyperparameters after testing out hyperparameter configurations on seven benchmark datasets. These importance scores represented the extent to which a hyperparameter impacts the accuracy of the wide-kernel CNN. In particular, the results identified three especially important hyperparameters: the number of filters in the first layer, the kernel size in the first layer, and the number of filters in layers 3-5.

In the aforementioned past work there was also an analysis of the influence each hyperparameter on other hyperparameters. The 'influence' of A on B was interpreted as the likelihood that the optimal value for B will change as a result of tuning A. In other words, A is influential on B if tuning A is likely to mean that B needs re-tuning.

When someone tunes a network, they tweak the options available to them until they find hyperparameter settings that result in better accuracy. One approach to the problem is to focus on a single choice – for example the number of convolutional filters used in the first layer of the network – and modify this value without changing anything else. We call this approach tuning a hyperparameter 'individually'. Tuning individually reduces the number of possibilities, perhaps to testing out a handful of values the hyperparameter might reasonably take. After trying the possibilities the best performer will be retained.

An underlying assumption of individually tuning hyperparameters is that there is a definite starting point. All of the other hyperparameters must be given a fixed value at the beginning of the process. The results from individual tuning are therefore contingent on there being a starting point and indeed we may find that a different starting point leads to a different optimal value for the hyperparameter being tuned. Even if a hyperparameter has been given the 'best' value through individually tuning it, we may find that the best value changes when another hyperparameter has been edited. In which case, the original hyperparameter needs to be re-tuned to reflect a new starting point.

It is not clear whether changing one hyperparameter will have any impact on another. It is possible for hyperparameters to be largely independent so that we can modify one of them freely without it affecting which value is optimal for the other hyperparameter. Alternatively, they could be closely linked and it might be best to tune them so they always change together. In our research we consider the derived question of whether tuning one hyperparameter individually has an impact on the outcome from tuning another hyperparameter. The answer will be a probability because we need to summarise what happens across a large range of starting points. We have seven hyperparameters and each way of setting them is a different starting point.

We therefore revisit the notion of 'influence' used in [1] and computed by Algorithm 1. This fairly straightforward method to quantify influence considers every combination of values in a grid search as a starting point. From each starting point the algorithm tests the influence of individually tuning one hyperparameter (let us call it 'A') on another hyperparameter ('B'). First, B is tuned, then A is tuned and subsequently B is re-tuned to see if the optimal value has changed. If re-tuning changes the value of B then the algorithm increments a running count. At the end, the algorithm returns the final value of the running count divided by the number of starting points that were tested.

When reporting these influence scores, [1] provided a visualisation to show the influence each hyperparameter has on one another. In the current paper, we repeat this procedure but consider what happens when only the three most important hyperparameters are tuned and the other hyperparameters are held to their default values (as defined in [12]). We show this new information since it allows us to focus on strategies that allow for faster and more effective tuning. Excluding hyperparameters that have low impact on the accuracy of the CNN makes it possible to cut down on computation.

3.3.4. Resampling

When tuning the first layer of the wide-kernel CNN we often find that the largest kernel size is best, but sometimes a small kernel is best and sometimes it does not matter. It depends what benchmark dataset is being used and therefore seems to be a result of differences in the data. We try to isolate

Algorithm 1 Compute the influence of A on B

```
trialCount $\leftarrow$ 0
differenceCount $\leftarrow$ 0
for each configuration $\mathbf{c} \in (\mathbf{c_1}, \mathbf{c_2}, ..., \mathbf{c_{12960}})$ do
    $B_{tuned} \leftarrow$ tune($\mathbf{c}$, $B$)
    $\mathbf{c}' \leftarrow \mathbf{c}$
    $\mathbf{c}'[A] \leftarrow$ tune($\mathbf{c}'$, $A$)
    $B_{re-tuned} \leftarrow$ tune($\mathbf{c}'$, $B$)
    trialCount $\leftarrow$ trialCount +1
    if $B_{tuned} \neq B_{re-tuned}$ then
        differenceCount $\leftarrow$ differenceCount +1
    end if
end for each
return differenceCount / trialCount
```

which dataset properties cause different kernel sizes in the first layer to be better or worse. This leads to two types of data manipulation, the first of which is resampling.

One practical question about how to tune the hyperparameters of a wide-kernel CNN is whether to resize the convolutional filters, particularly in the first layer, when the sampling rate changes. Fortunately, the sampling rate is something that we can control artificially. We resample windows of vibration data to a lower rate, for example going from an initial window sampled at 48 kHz to a window of data which now contains the same content but is sampled at 24 kHz instead. The highest frequencies will be lost, but otherwise the signal will be preserved with high accuracy. Simply it will take fewer data points to record the same back-and-forth vibrations, and each data point will represent a bigger jump forward in time from the previous one.

To investigate the impact of the sampling rate on hyperparameters such as the kernel size of the first layer, we make multiple copies and resample them by a factor of 2, 4, 8 and 16. Starting off with data sampled at 48 kHz, this results in versions sampled at 24 kHz, 12 kHz, 6 kHz and 3 kHz. On each version of the data, we run a grid search over the hyperparameters which previous work showed were the most important. The grid search results can be analysed to compare what hyperparameter settings work best at different sampling rates.

For each combination of possible hyperparameters in our grid search, we train and test on the original and the four resampled versions of the CWRU 48 kHz dataset. The CWRU data was chosen because it combines: a relatively high sampling rate, a well-documented and high-quality data collection procedure, remaining a moderate challenge for neural networks, and being perhaps the most widely used benchmark in the field.

3.3.5. Filtering

It has been suggested in past work that the reason a wide kernel boosts the accuracy of a neural network is that wide kernels are good at filtering out high-frequency noise. We investigate this by employing a lowpass filter to deliberately remove high frequencies from the data. If the main accomplishment of a wide kernel is to filter out high frequencies, then a larger range of kernel sizes should become viable if the high frequencies are already eliminated by pre-filtering. We do not know what (if any) cutoff should be used to decide which frequencies to keep and which to throw away, so we try multiple cutoff thresholds, starting from near the upper limit provided by the Nyquist frequency. In real terms, given data like CWRU sampled at 48 kHz, we apply filtering to multiple copies of the data. We filter with cutoff frequencies of 12 kHz, 6 kHz, 3 kHz, 1500 Hz, 750 Hz, 375 Hz, 187 Hz, 93 Hz and 46 Hz. An additional benefit of using many cutoff thresholds is that we can also check if any of them are too drastic, shedding light on which frequencies are essential for accurate fault detection and cannot be removed.

It should be noted that filtering in this way does not impact the sampling rate or the size of a window of vibration data, it simply removes content which occurs at high frequencies. This makes it easy to compare the results when performing a grid search over the most important hyperparameters. A grid search can be performed on each filtered version of the CWRU dataset, and the results are directly comparable since the same architectures are applied to data which has exactly the same size and shape.

3.3.6. Analysis Techniques on Resampled and Filtered Data

We analyse the resulting information in multiple ways. First, we look at correlations between different versions of the dataset, which gives an indication of the impact that resampling had. If resampling had no impact at all, then we would expect to have a perfect correlation of 1.0, and values lower than this would imply that resampling has an influence on which hyperparameter combinations are stronger and which are weaker.

Afterwards, we look at how important different hyperparameters are. We do this by initially asking a related question of whether the test accuracy of a network can be predicted from the hyperparameters it has. If the accuracy can be predicted, we can hone in on the specifics of how much each hyperparameter contributes to the predictability.

Correlation analysis: By resampling the CWRU data, we create multiple versions of the dataset. One obvious question that arises is whether a combination of hyperparameters that works well on one version of the data will also work well on another. A similar question could be asked about the worst-performing hyperparameters. In general, this is a question of correlation: to what extent do the accuracy scores obtained on one dataset version correlate to the scores on data resampled to a different rate? We compute and then display the correlations, which gives an insight into how much the hyperparameters are affected by the resampling/filtering. One possibility is that certain hyperparameter combinations are simply better than others, and it does not matter if the properties of the data (such as sampling rate) change. In the extreme, this would result in a correlation of 1.0 between all versions of the dataset. However, if smaller correlations are found, then it would suggest that the properties of the data impact which hyperparameters are best, or that there is some degree of randomness in the accuracy scores that deflates the correlations.

Box plots: One of the key questions in our research is whether the optimal kernel size in the first layer changes when the data is either resampled or filtered. Box plots provide a straightforward view onto how well different kernel sizes tend to work. We plot the distribution of accuracy scores for each kernel size when applied to each version of the data.

Feature importance based on Shapley values: After performing a grid search over different values that the hyperparameters can take, we are left with a set of accuracy scores. Each entry in the results records the specific values that were used for the three hyperparameters manipulated in the grid search, plus the trained network's accuracy on test data. We use these four pieces of information to investigate the relative importance of each hyperparameter. The analysis proceeds by analysing one grid search at a time, before comparing across grid searches. Remember that a grid search represents a certain level of resampling or filtering, e.g. it tells us what happens when different hyperparameter values are tried out on CWRU data resampled from 48 kHz to 24 kHz. Another grid search covers CWRU data resampled from 48 to 12 kHz, and so on.

'Importance', of course, can mean many things. Here, we provide one particular approach to quantifying the importance of hyperparameters. Our approach asks how much each hyperparameter seems to contribute to the final accuracy of a network. If a hyperparameter is important, then setting it correctly tends to result in a noticeably higher accuracy. If a hyperparameter is not important, then even changing it drastically will not cause any increase or decrease in how accurate the network is.

It is difficult to directly state how much a hyperparameter influences the accuracy of the network because this can be contingent based on other hyperparameters or could be subject to a nonlinear relationship which is hard to identify using basic statistical analysis. Our past work showed that *nonlinear* regression methods were noticeably better at the task of predicting how accurate networks will be. In order to model the obscure, nonlinear relationships between hyperparameters and accuracy, we use a multilayer perceptron. It is a simple neural network – ours contains two hidden layers of 32 neurons – which is told the how the hyperparameters of a wide-kernel CNN have been configured and is trained to predict what accuracy those hyperparameters lead to. The outputs from grid searches provide the data used for training the MLP.

Once the MLP is trained, it contains information about the importance of different hyperparameters. In order to probe the MLP further we train versions which have access to a hyperparameter (i.e., are told what value the hyperparameter has) and versions which do not. If the versions without access to a hyperparameter perform much worse then the hyperparameter is seen as important. Shapley values provide a framework for fairly combining information from all the possible permutations of inclusions/exclusions of hyperparameters [84]. By visualising these Shapley values, we can look into the relative importance of each single hyperparameter, measured in terms of how much it contributes to reducing the MLP regressor's Mean Absolute Error (MAE). This is the method we employed earlier in [2], although there we applied it to raw benchmark data rather than filtered and resampled CWRU data.

Calculating the Shapley value of a hyperparameter is done by permuting the order of all hyperparameters and then performing two regressions (using the MLP): one with the current hyperparameter included and one without.

The difference in error between these two regressions indicates the hyperparameter's contribution to the overall model's performance. By averaging over all possible permutations, we obtain the precise Shapley value. For an approximation, a set number of permutations can be chosen at random. For the sake of computational efficiency, we use 100 random permutations. As seen in [85], Shapley values tend to converge quite quickly to a reasonable approximation, and so we do not feel it is necessary to run all permutations. Following the notation in [85], the calculation can be written down follows:

$$\varphi_i = \frac{1}{|N|!} \sum_{\mathcal{O} \in \pi(|N|)} [v(Pre^i(\mathcal{O}) \cup \{i\}) - v(Pre^i(\mathcal{O}))] \tag{3}$$

where φ_i is the Shapley value of feature i, $|N|$ is the number of features, $\mathcal{O}$ is a randomly-chosen permutation of the features, $v(s)$ is the average test performance (mean absolute error or MAE) of a regression using only the set of features s, and $Pre^i(\mathcal{O})$ is the set of features that appear earlier than feature i in the permutation $\mathcal{O}$.

Feature importance based on d-dimensional earth mover's distance: We want to understand how much the accuracy of the neural network changes depending on specific settings of the hyperparameters. To do this, we first look at how to summarise a single setting of a single hyperparameter, such as giving the filter in the first layer a width of 256. When the first layer has a filter of width 256, many possibilities remain for the other hyperparameters. Within our grid search, all the possibilities are exhaustively tested, the upshot of which is that we have a collection of many tested networks that all have a first layer filter width of 256. The test accuracy scores from a collection of networks, taken together, form a distribution. Some accuracy scores will be more frequent than others.

The idea of our analysis is to take different distributions and compare them. We can compare the distribution of accuracy scores when the first layer has a filter size of 256 to when the filter size is 128, for example. If two distributions are wildly different, it clearly shows that changing the hyperparameter has caused a change in what accuracy you can expect from the network.

To summarise this information, we use a metric called the d-dimensional Earth Mover's Distance (EMD) for each hyperparameter [86]. The EMD essentially measures the difference between two probability distributions, reflecting how much one needs to be adjusted to become the other. Higher EMD values indicate greater dissimilarity. The EMD is the same as the Wasserstein difference when comparing two distributions, but it can be extended to compare more than two distributions (hence 'd-dimensional'). Analysing all the values that a hyperparameter can take, we come up with a single EMD that summarises the hyperparameter overall. With EMD, we can assign a single number to each hyperparameter, representing its impact on accuracy scores. This is a method we used earlier in [1], although there we applied it to raw benchmark data rather than filtered and resampled CWRU data.

It is important to note that some hyperparameters are tested with more values than others are. For instance, we might test five kernel sizes but only three strides and this makes the EMD values somewhat inconsistent and difficult to interpret. To make the results more usable we add an extra processing step that begins by calculating a baseline EMD for each hyperparameter. We do this by calculating the EMD for surrogate data in which the accuracy scores have been shuffled and therefore the relationships between hyperparameters and accuracy have been destroyed. Shuffling multiple times and recalculating the EMD allows us to estimate the average EMD due to chance. When we present the EMD for each hyperparameter we make the numbers more comparable and meaningful by dividing by the baseline.

3.4. Summary of Datasets Used

Across the various parts of our research, multiple datasets are employed. For the sake of clarity we now present the full collection of datasets we use and remark on which architectures they are used with. Table 4 shows these details.

4. Results

Here we analyse the outputs from our grid search over hyperparameters, which has been applied to multiple datasets and to multiple filtered and resampled versions of the CWRU dataset. Combining these analyses leads to a broader understanding of how to set hyperparameters under different conditions. Before doing that however, we first attempt to establish the importance of hyperparameter tuning outside the context of a wide-kernel CNN by looking at how alternative architectures perform while their hyperparameters change.

Dataset	**Used With**
Unmanipulated benchmarks	
CWRU	LSTM, Transformer, Wide-kernel CNN
Gearbox	LSTM, Transformer, Wide-kernel CNN
MFPT	LSTM, Transformer, Wide-kernel CNN
Paderborn	LSTM, Transformer, Wide-kernel CNN
SEU	LSTM, Transformer, Wide-kernel CNN
UOC	LSTM, Transformer, Wide-kernel CNN
XJTU	LSTM, Transformer, Wide-kernel CNN
Resampling	
CWRU 48 kHz resampled to 24 kHz	Wide-kernel CNN
CWRU 48 kHz resampled to 12 kHz	Wide-kernel CNN
CWRU 48 kHz resampled to 6 kHz	Wide-kernel CNN
CWRU 48 kHz resampled to 3 kHz	Wide-kernel CNN
Filtering	
CWRU 48 kHz lowpass at 12 kHz	Wide-kernel CNN
CWRU 48 kHz lowpass at 6 kHz	Wide-kernel CNN
CWRU 48 kHz lowpass at 3 kHz	Wide-kernel CNN
CWRU 48 kHz lowpass at 1500 Hz	Wide-kernel CNN
CWRU 48 kHz lowpass at 750 kHz	Wide-kernel CNN
CWRU 48 kHz lowpass at 375 kHz	Wide-kernel CNN
CWRU 48 kHz lowpass at 187 kHz	Wide-kernel CNN
CWRU 48 kHz lowpass at 93 kHz	Wide-kernel CNN
CWRU 48 kHz lowpass at 46 kHz	Wide-kernel CNN

Table 4: Summary of datasets used in the research.

4.1. Results Pt. 1 – Experiments on Multiple Architectures

The overall focus of our work is a specific wide-kernel CNN architecture. However, it is important to note that other architectures also present the user with hyperparameter decisions, and also suffer from the fact that poor hyperparameter choices can be disastrous for accuracy. We therefore briefly present some experiments with two other popular architectural styles: a recurrent LSTM type of neural network and a transformer network. These results establish the general importance of hyperparameter tuning when working with neural networks, which is the basis for the remaining sections where we deal with the wide-kernel architecture much more extensively.

4.1.1. Long Short Term Memory (LSTM)

Across our sample of 100 LSTM configurations, we see that poor hyperparameters can be disastrous. In the worst cases (CWRU and XJTU) the lowest-performing network was wrong more than 10 times as often as the best network. On all datasets the default values do roughly as well as about 75% of modified hyperparameter settings and therefore are reasonable choices. One interesting thing is that the LSTM network's default values appear to under-perform on XJTU data. Facing this benchmark, an LSTM configured with the defaults only achieved an accuracy of 72% whilst it was possible to get above 90% with other hyperparameter values. Re-tuning the default hyperparameter settings seems to be a good idea on XJTU data, indicating that hyperparameters that perform well on one dataset do not necessarily perform well on another dataset. The default values fared relatively much better on CWRU and SEU. Overall the results show that hyperparameter tuning has a noticeable impact on the LSTM architecture much like with the transformer and wide-kernel CNN networks.

	Min	25%	Median	75%	Max	w/ Defaults
CWRU	35	59	68	80	94	81
Gearbox	6	9	11	13	17	14
MFPT	46	49	50	50	54	52
Paderborn	28	40	46	50	57	48
SEU	62	88	96	98	99	94
UOC	24	35	41	50	72	54
XJTU	26	48	57	75	97	72

Table 5: Tukey's five-number summary of how accurate the LSTM architecture was with different hyperparameter settings. The accuracy score when using the default hyperparameter values is also shown in the right-hand column.

4.1.2. Transformer

The transformer architecture performs relatively well when using the default hyperparameter settings, as shown in Table 6. On four of the baselines it was able to correctly identify fault conditions on more than 90% of test cases. Looking across all of the baseline datasets, the default hyperparameter settings were at least as good as 75% of altered versions of the settings. The results suggest that the default values are quite reasonable choices. On the Gearbox dataset the transformer was inaccurate no matter if the default hyperparameter settings were used or not, suggesting a specific problem with that one dataset. Importantly, our results show overall that the transformer architecture is sensitive to hyperparameter tweaking. On CWRU and UOC for example, the accuracy plunges by roughly 90% when swapping the best-performing hyperparameters for the worst.

Overall, the transformer architecture beats the LSTM architecture so long as it is given the right hyperparameter settings. The transformer was more sensitive to changes in the hyperparameters our study explored, in the sense that it could be very accurate or be very inaccurate depending on whether those were right. However, getting the hyperparameters right paid off for the transformer since its max accuracy was 40% higher than the LSTM's max when testing on UOC data. Given the correct hyperparameter settings, the transformer could be at least as good as any LSTM network we tried out on the seven benchmarks.

	Min	25%	Median	75%	Max	w/ Defaults
CWRU	8	16	69	89	100	93
Gearbox	4	5	11	14	24	13
MFPT	18	46	62	68	77	68
Paderborn	10	26	54	62	68	62
SEU	20	23	87	96	100	95
UOC	8	17	82	95	100	94
XJTU	7	35	89	95	99	97

Table 6: Tukey's five-number summary of how accurate the transformer architecture was with different hyperparameter settings. The accuracy score when using the default hyperparameter values is also shown in the right-hand column.

4.2. Results Pt. 2 – Focusing on the Wide-Kernel CNN Architecture

The main body of our research is about the wide-kernel CNN architecture described in Section 3.3.1. We begin the results by revisiting how successful networks in a grid search are when trained and tested on seven different benchmark datasets. This grid search is the same as was analysed before in [1], although here our results have a different focus. We look at the average test results when setting individual hyperparameters to specific values, and we also zoom in on the interactions between the hyperparameters that were previously identified as most important. These results help us formulate guidance on how to tune the hyperparameters of the wide-kernel CNN, in later sections of this paper.

After revisiting the grid search applied to seven benchmark datasets we shift our attention to how well wide-kernel CNNs work on manipulated versions of the CWRU data. Deliberately manipulating the data is a means to get greater control of data properties and see how they influence which hyperparameter values are best. We maintain a special interest in the kernel size in the first layer (the 'wide kernel' of the wide-kernel architecture) because this seems to be the most data-dependent hyperparameter. The other hyperparameters of the network appear fairly consistent in terms of what is the correct way to tune them, but the first layer kernel size sometimes needs to be shrunk and sometimes needs to be expanded. By first resampling and then filtering we look at whether the sampling rate and the spectral content have an impact on what kernel size is best.

4.2.1. Summarising Grid Search Results

Now we look at how the accuracy of the wide-kernel CNN varies as we change both the hyperparameter values and the data. Each condition's accuracy is shown in Table 7. These results are based on the same grid search used previously in [1], which trained and tested different combinations of hyperparameters on 7 benchmarks. We revisit the grid search in order to pull out and highlight helpful information when thinking about how to tune the hyperparameters on different datasets.

First, we see that the accuracy tends to be higher on some benchmark datasets than others. It seems that some tasks were more difficult or were more demanding within the constraints imposed by the grid search. Differences are not unexpected however, since each dataset is distinct in terms of recording procedure and number and type of faults.

Perhaps more interesting are the individual accuracy scores showing what happens when we tune different hyperparameters. The kernel size in the first layer reveals the most unexpected results. With CWRU, Paderborn and UOC there is a strong effect whereby a longer kernel does better and the longest kernel exceeds the shortest by at least 15% accuracy. The MFPT and XJTU benchmarks exhibit a weaker effect of roughly 6% difference between the longest and shortest kernel. The kernel size in layer 1 barely effects the results on Gearbox data. Meanwhile, a strong contradictory effect is observed when using SEU data. The shortest kernel is much better (more

Hyperparameter	Value	CWRU	Gearbox	MFPT	Paderborn	SEU	UOC	XJTU
Kernel size layer 1	16	25	78	47	74	87	32	64
	32	31	78	48	84	80	35	59
	64	40	77	49	87	70	40	64
	128	55	77	51	89	61	49	64
	256	69	77	53	89	53	58	70
Stride layer 1	4	41	77	48	79	69	35	54
	8	44	80	50	86	70	45	66
	16	47	75	51	89	72	49	73
Filters layer 1	8	43	74	53	84	74	57	69
	16	53	76	56	86	74	61	68
	32	53	77	54	85	72	51	65
	64	45	77	49	85	70	37	63
	128	37	79	45	84	67	28	61
	256	33	81	42	83	65	23	60
Kernel size layer 2	3	43	78	50	85	71	43	67
	6	45	77	49	84	70	43	62
Filters layer 2	8	44	78	52	85	74	49	68
	16	50	78	52	85	73	50	66
	32	51	79	52	85	71	47	63
	64	47	77	50	85	69	42	62
	128	40	76	47	85	68	37	63
	256	33	76	45	84	66	33	65
Kernel size layers 3-5	3	45	80	50	84	71	43	65
	6	43	75	49	85	69	42	64
Filters layers 3-5	8	33	83	49	86	67	49	75
	16	45	91	50	89	72	51	82
	32	53	90	51	89	73	48	75
	64	52	81	50	88	73	41	64
	128	46	67	49	82	70	36	52
	256	35	52	49	74	65	32	39

Table 7: The average performance for different values of wide-kernel CNN hyperparameters when trained and tested on different datasets.

than 30% better) compared to the longest kernel size. In all, it seems that longer kernels in the first layer tend to be best but it very much depends on the fault vibration data.

The stride in the first layer is a simpler story. In the case of Gearbox it does not affect the accuracy by more than 5% whilst in all the other datasets it is clear that the highest value (16) is best. A greater stride reduces computation by shrinking the outputs and potentially removes redundancy by decreasing the overlap of the outputs.

The number of filters in the first layer was also identified as important by past work. Good performance is often achieved when there are 16 or 32 filters in the first layer. Either 16 or 32 was the best on every dataset except XJTU (when 8 filters was 1% better) and Gearbox. With Gearbox, the trend is that more filters gives higher accuracy and 256 beats 8 filters by about 7% accuracy. Looking across all the datasets, 16 or 32 are the safest bets when tuning this hyperparameter but exceptions are possible.

The final ‘important’ hyperparameter is the number of filters in layer 3 to 5. We observe that either 16 or 32 works best on all the benchmark datasets. This conclusion at least is straightforward.

4.2.2. Influence of Important Hyperparameters on One Another

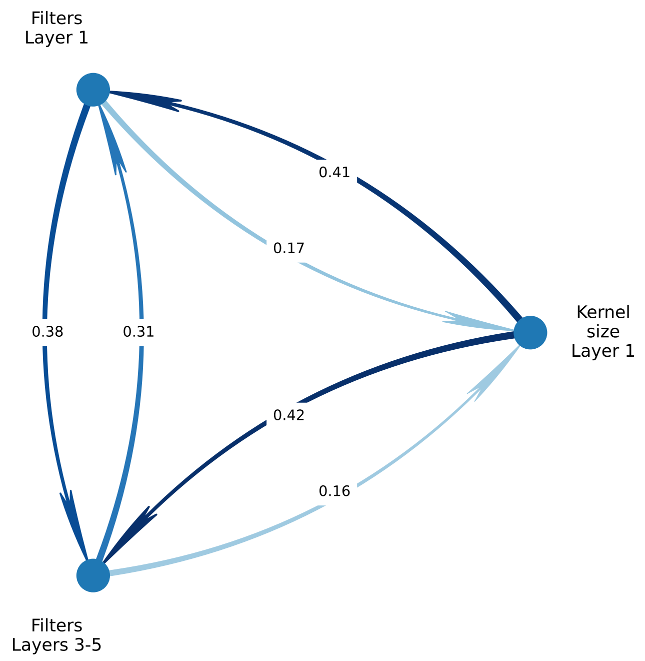

Figure 6: The likelihood that tuning one hyperparameter (source of an arrow) will cause another (destination of an arrow) to need re-tuning.

We now take another deep dive into the three hyperparameters identified as important by our previous work. It is very computationally expensive to run a full grid search over all the possible hyperparameter combinations and so someone applying the network to new data might well want to know how to tune the hyperparameters one-at-a-time. We ask which is the best order to tune the three most important hyperparameters sequentially. The method we used in our previous paper [1] comes in handy again here: we look at how much one hyperparameter influences another hyperparameter which has already been tuned. Because the hyperparameters are interconnected it is possible that a hyperparameter that started off being optimal no longer has the best value when a different hyperparameter changes. We look at how often this phenomenon happens and we summarise the information as an 'influence' score. If A is highly influential on B then changing A most likely means changing the best value for B. We can see the numbers in Figure 6.

The kernel size in layer 1 appears to be the most influential and therefore it appears to be the best one to tune first. Tuning it later on would likely disrupt the other hyperparameters and cause them to need re-tuning. In other words, there is often no point in tuning the number of filters in layers 1 or 3-5 beforehand, since they will need to be tuned again when the kernel size changes. After choosing the size of filters in the first layer it is best to choose how many next. The choice of how many filters in layers 3 to 5 can be delayed to the end.

4.2.3. Resampling

The results of a grid search over the three most important hyperparameters, performed once each for the different resampling conditions, are brought together in the results here. Visualising the results makes it possible to spot any obvious trends in how the (increasing levels of) resampling impact the performance of different hyperparameter settings. We present several analyses on both CWRU 48 kHz data to see how much each hyperparameter seems to influence performance. These analyses include using a method based on Shapley values and using a method based on d-dimensional earth mover's scores to pinpoint important hyperparameters. Here, we include the results obtained when resampling CWRU data, starting from data originally sampled at 48 kHz.

First, we consider how much the accuracy scores change as a consequence of resampling the data. Correlations tell us whether the best-performing combinations of hyperparameters on one version of the data also remain the best after resampling, and vice versa with the worst performers. The precise correlation results are shown in Figure 7. All correlations are positive which implies that the hyperparameter values that do well on one version of the data have a tendency to perform well on data resampled to a different rate. The weakest correlation is between the original 48 kHz data and the most

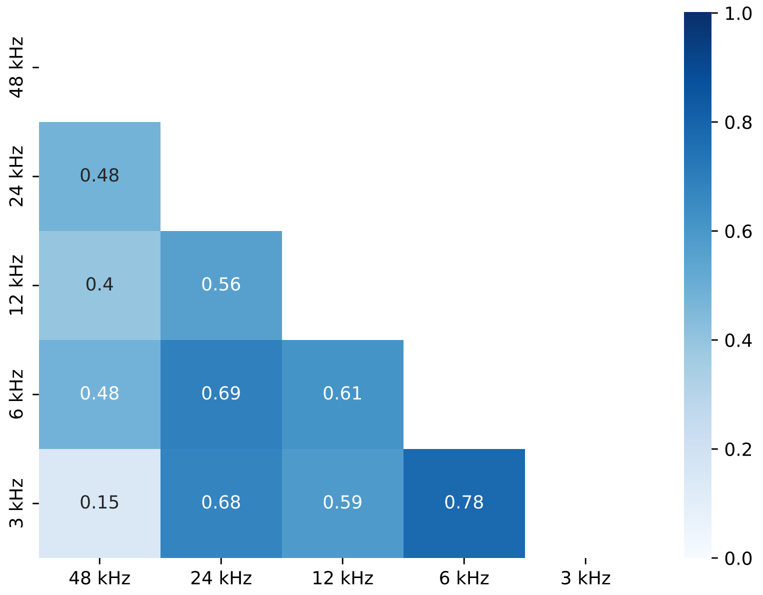

Figure 7: Correlation between different resampling conditions, starting from 48 kHz.

resampled data of 3 kHz. This is not surprising since they are the furthest away from each other in terms of data properties. The weakened correlation between 48 kHz and 3 kHz gives us an initial

Before moving on we must make a brief disclaimer about the correlations, in particular we acknowledge there is a ceiling effect because many networks achieve close to the maximum accuracy of 1.0, and, the training and testing of neural networks is stochastic and so there may be some random noise in the accuracy scores. There is a chance that these statistical quirks have inflated or deflated the correlations, and so we do not report p-values which might be biased by these effects. Instead, we view the results as indicative. The results give an indication that hyperparameter settings that are stronger on one version of the data are more likely to be stronger when also used on a resampled version of the data, but room is also left for differences between sampling rates to emerge.

We take a closer look at the kernel size hyperparameter in particular since we want to know if the kernel size needs to change when the sampling rate is altered. The box plots in Figure 8 let us see more clearly which specific kernel sizes are best and not only how important the hyperparameter is overall.

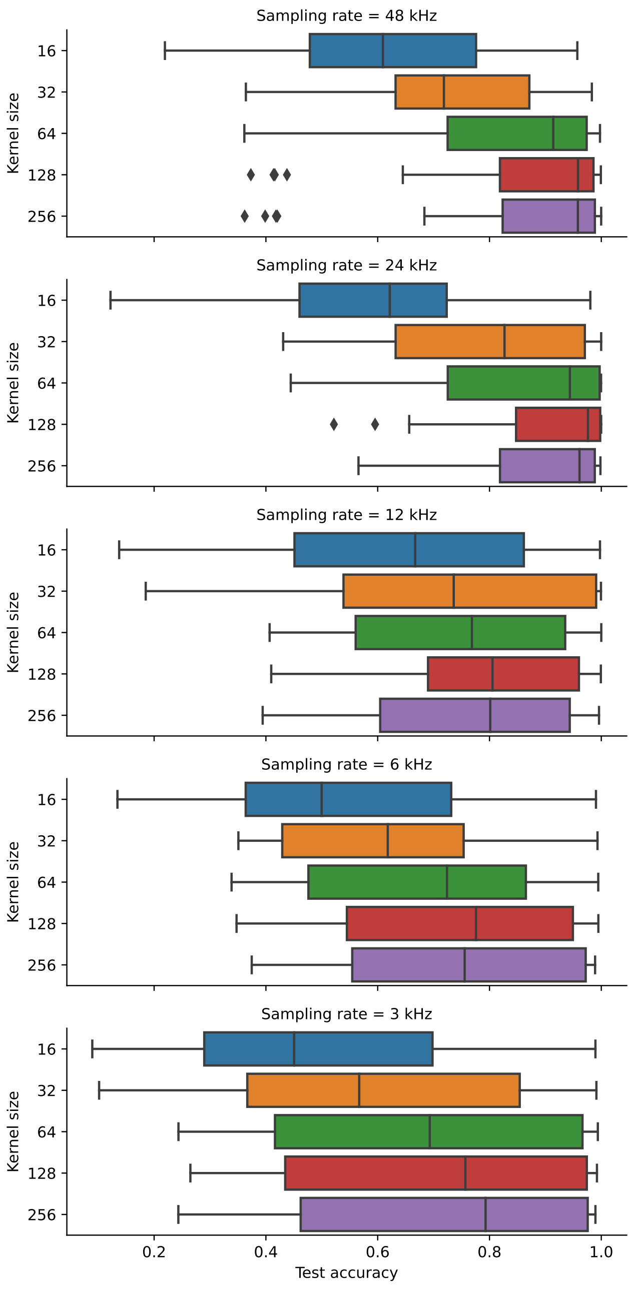

Figure 8: Accuracy scores for different kernel sizes in the first convolutional layer, as tested on different resampling conditions for data originally sampled at 48 kHz.

These plots confirm that larger kernels perform better, with 128 or 256 being the best even when resampling from 48 kHz to 3 kHz. Resampling condenses the data and makes the inputs shorter and one might expect smaller kernels to become more effective as a result of increased resampling. The results however show no such trend. When looking at the box plots we also observe that the task becomes more difficult as the data is resampled further. Either the loss of information that comes with compression or the impact of reducing the length of inputs could explain the downwards trajectory. Considering the results in Figure 8, resampling to 24 kHz at least seems to have no negative impact and so it could be unnecessary and wasteful to record at 48 kHz when a lower sampling frequency is equally good. Further research would be needed to establish this more firmly.

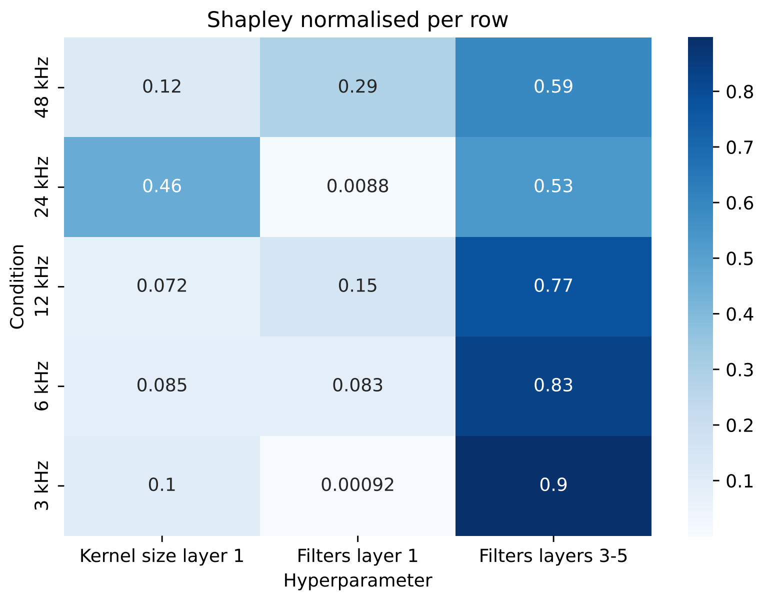

Figure 9: Feature importance after resampling to different rates, starting from 48 kHz. Each row is normalised to show the relative importance of the hyperparameters.

In the next piece of our analysis we look at feature importance. This shows us the impact of resampling on the other two most important hyperparameters alongside kernel size in the first layer. We interpret importance by saying that if a hyperparameter is important then it determines whether a neural network is accurate or not. If a hyperparameter is important and you know its value, then you can predict the accuracy of the network. We arrive at numeric importance scores by looking at the contribution (Shapley value) of each hyperparameter to the successful predictions of accuracy of a multi-layer perceptron. Figure 9 shows the Shapley values after normalising each row to add up to 1.0. Naturally, higher values indicate more importance.

No matter the degree of resampling, the number of filters in layers 3-5 is the most important. Interestingly, this hyperparameter is increasingly dominant as the resampling gets more extreme. With no resampling or with resampling to 24 kHz, the other hyperparameters constitute almost half of the Shapley values. There is no obvious pattern to the kernel size, and the much higher importance when resampling to 24 kHz is an unexplained outlier. In any case, the kernel size did not keep on becoming more important as a result of escalating resampling. Instead, the clearest conclusion is that the filters in the later layers become more important when resampling is greater. If we consider that the first layer is often used to process raw, noisy signals then the declining importance of this layer might reflect the fact that more resampled data is already more processed before it reaches the neural network as an input. Alternatively, networks' overall lower accuracies on highly resampled data (shown in Figure 8) present the possibility that resampling deletes patterns in the signal, hence the first layer is important when processing mildly resampled data since it allows networks to pick up on patterns useful for successfully classifying vibrations.

Another way to consider how important different hyperparameters are is to ask how much the accuracy scores change when the hyperparameter is changed. The d-dimensional earth mover's distance (EMD) quantifies just this. Figure 10 shows how many times larger the observed EMD is than a surrogate baseline. Firstly we see that compared to baseline values, the number of filters in the first layer has little impact. The number of filters in layers 3-5 however are much more important no matter what resampling has been applied. The size of the filter in the first layer is interesting because it seems important with no resampling or little resampling, but becomes unimportant when resampling to 12 kHz or less.

4.2.4. Filtering

In this section, instead of resampling we look at filtering the data. Filtering modifies the information that is included in the data such that certain frequencies are removed, whilst holding constant the sampling rate and the time series length. This gives us some companion results to the resampling

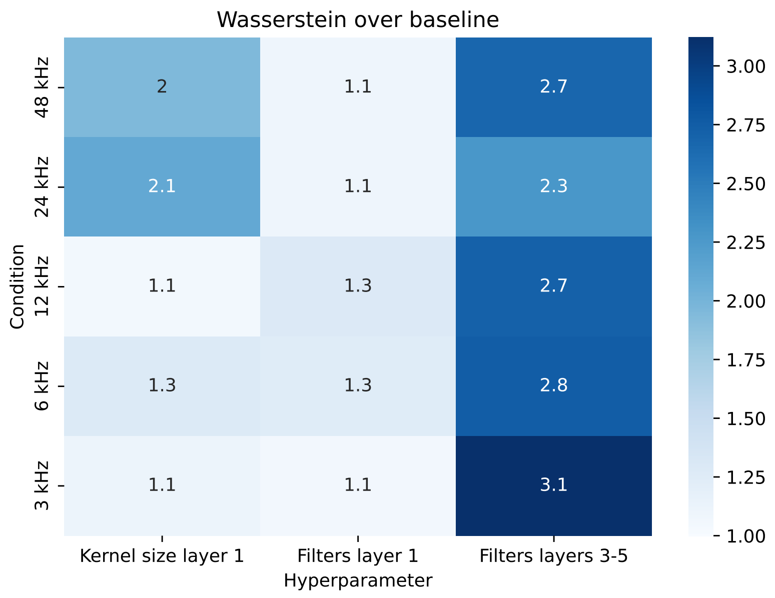

Figure 10: How much the accuracy scores change as a result of tweaking a hyperparameter, when processing data with different levels of resampling, starting from 48 kHz. Each cell is normalised to show its value relative to surrogate data (see explanation in Section 3.3.6).

results we just showed. The filtered data isolates the impact of the content of the signals whereas the previous resampling also made the sequences shorter and more condensed.

When processing data sampled at 48 kHz we have nine filtering conditions: low-pass with a threshold of 12 kHz, 6 kHz, 3 kHz, 1.5 kHz, 750 Hz, 375 Hz, 187 Hz, 93 Hz and 46 Hz. These can be compared side by side. We use the methods described earlier, namely employing Shapley values and the d-dimensional earth mover's scores, to investigate the impact of filtering. It had been suggested that a wide kernel is beneficial primarily because it filters out high-frequency noise. The filtering experiments make it possible to verify whether this really is the benefit granted by a wide kernel. Here, we include the results obtained when filtering CWRU data, applying different thresholds to decide how many frequencies to filter out.

Correlations between the different conditions appear in Figure 11. We see that many levels of filtering, especially the weakest forms that only remove the highest frequencies, give correlations in the range 0.5 to 0.8. These quite high correlations imply that moderate filtering did not have a large impact

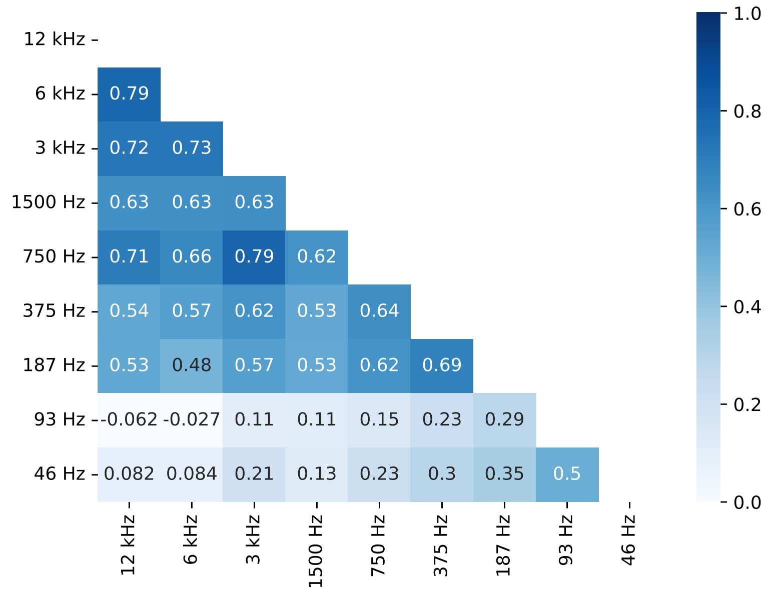

Figure 11: Correlation between filtering conditions with 48 kHz CWRU data.

on how to set the hyperparameters. Filtering at a threshold lower than 187 Hz however causes a sudden drop in correlations. The most extreme forms of filtering evidently do affect which hyperparameter settings work well. We observe that the average accuracy is much lower when filtering at 46 or 93 Hz (the average accuracy was 0.29 in both cases) than when filtering at 3000 Hz or above (the averages ranged from 0.95 to 0.97), and there might simply be no distinction between good and bad hyperparameter settings when using extreme filtering. If there are no good and bad hyperparameter settings then the results will be more influenced by noise and will correlate less, which is one possible explanation for the relatively low correlations seen. In summary, hyperparameters generalise fairly well and so the same settings do well on multiple levels of filtering, except when the strongest filtering is applied. Filtering changed the data properties but did not completely redefine which settings are good or bad.

In order to make a deeper dive into the kernel size in the first layer we can plot the accuracy scores visually for each kernel size. Figure 12 shows the spread of accuracy scores obtained when using different lengths of kernel and when processing data with different levels of filtering. If the filtering

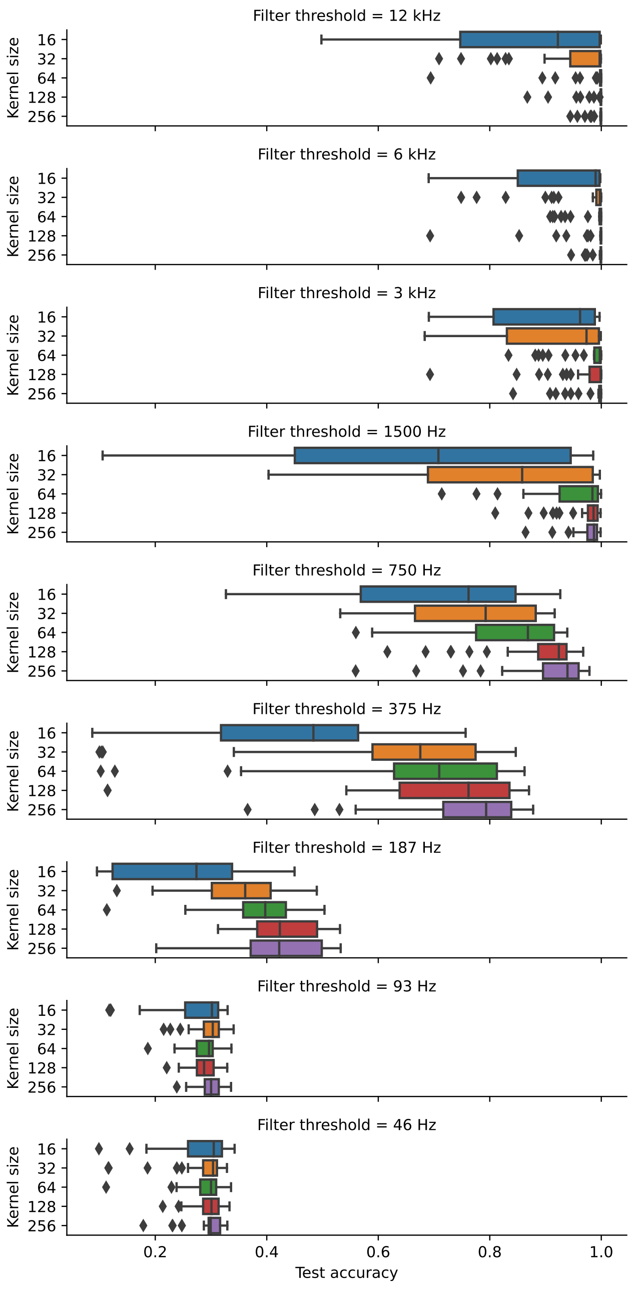

Figure 12: Accuracy when using different kernel sizes in the first layer, as applied to different filtering conditions when using data recorded at 48 kHz.

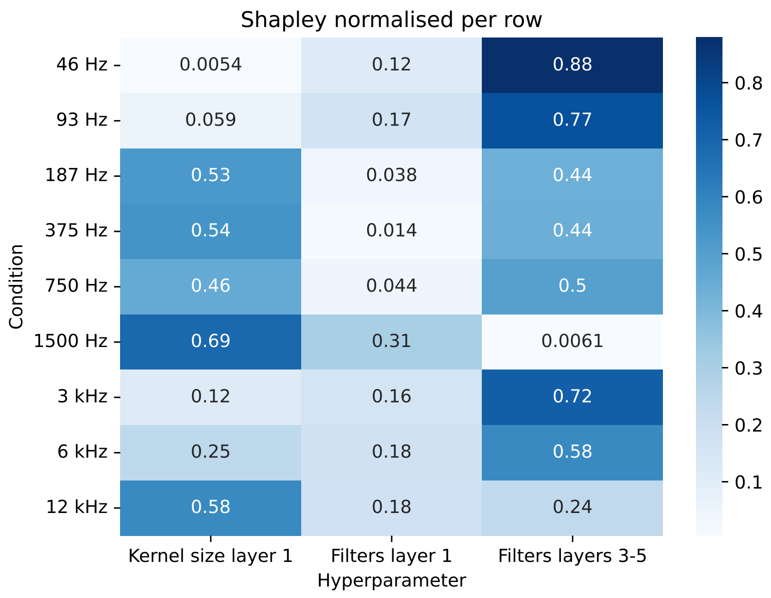

Figure 13: Feature importance after filtering 48 kHz CWRU data at different cutoff thresholds. Each row is normalised to show the relative importance of the hyperparameters.

is anything milder than 93 Hz we see that a larger kernel performs better. The value of a wide kernel does not relate to high-frequency noise as was suspected because any such noise would be removed by the filters. It seems that the benefit of a wide kernel must be explained in some other way.

We derive Shapley values that represent how much each hyperparameter helps when predicting how accurate a network will be. These importance values are large when a hyperparameter really tells us something about how well a network will perform. Figure 13 shows Shapley values derived from networks trained to classify faults from vibration data with different levels of filtering. Both the size of the first filters and the number of filters in the later layers are important. Note that the balance of importance between these two hyperparameters changes as filtering is increasingly applied but it changes without any obvious pattern.

The earth mover's distance (EMD) also paints a picture of how important different hyperparameters are. We show the EMD for our hyperparameters in Figure 14. This again confirms that the kernel size in the first layer and the number of filters in layers 3-5 are both important hyperparameters. The

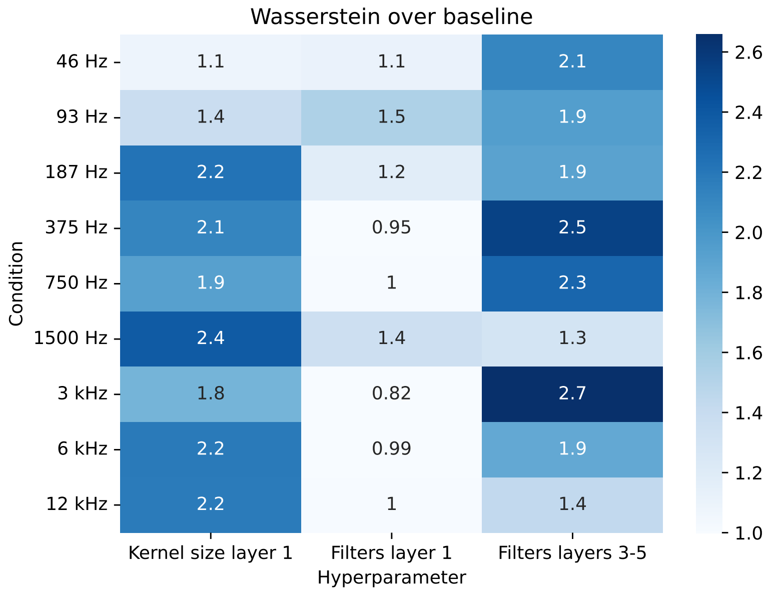

Figure 14: How much the accuracy scores change as a result of tweaking a hyperparameter, when processing 48 kHz CWRU data that was filtered at different cutoff thresholds. Each cell is normalised to show its value relative to surrogate data (see explanation in Section 3.3.6).

number of filters seems to become slightly more important when the filtering is increased.

5. Discussion: Hyperparameters of a Wide-Kernel CNN Explained

Having thoroughly analysed the hyperparameter space for our wide-kernel CNN we can now lay down some guidelines. We researched architectural hyperparameters such as the size of convolutional kernels in different layers and now we are ready to give the following advice. First, we note that some hyperparameters are less important and it should be possible to set these low in order to save computation. The kernel size and number of convolutional filters in the first layer are more important however and so we took additional care when analysing these. The number of filters in layers 3-5 is also important. We observe that larger kernels do better in the first layer and so suggest using a kernel of size 256 but we found that this depends on the dataset. Testing out different numbers of filters in layers 3-5 is a good way to improve performance although we see that a default value of 32 does well on multiple datasets.

Considering all of the above, we are then able to give fairly specific guidance on how to set the hyperparameters of the network. Going through the hyperparameters in turn:

- **First layer kernel size**: As a first attempt, it seems best to set this high and therefore use a wide kernel. Of all options we found that the highest value of 256 did the best usually. However, we also found that this hyperparameter is highly sensitive to differences between the datasets and so it should be carefully tuned if the initial accuracy of the network seems to leave room for improvement.

- **Stride in the first layer**: This does not greatly impact the accuracy so it may as well be set to the most computationally efficient value, which is actually the highest value since a long stride allows the network to skim faster over the inputs. In our experiments, that was 32.

- **Number of filters in the first layer**: We considered 8, 16, 32, 64, 128 and 256 as options and found that 16 and 32 are the best. We recommend 16 since it balances accuracy and computational efficiency. Although our results suggest it will not be necessary in most cases, since this is an important hyperparameter the user might decide to search for the optimal value amongst all the options.

- **Second layer kernel size**: This hyperparameter does not greatly impact the accuracy so it may as well be set to the most computationally efficient value. In our experiments, that was 3.

- **Number of filters in the second layer**: This hyperparameter also does not greatly impact the accuracy and can be set to the most computationally efficient value, which was 8 in our experiments.

- **Kernel size in layers 3 to 5**: This is another hyperparameter that does not greatly impact the accuracy and may as well be set to the most computationally efficient value. In our experiments, that was 3.

- **Number of filters in the second layer**: We considered 8, 16, 32, 64, 128 and 256 as options and found that 16 and 32 are the best. We recommend 32 as a reasonable default value. Although our results suggest that it will not be necessary in most cases, since this is an important hyperparameter the user might choose to search for the optimal value amongst all the options.

Our experiments suggest that if we use the default values recommended above, on average we would obtain an accuracy better than 75% of alternatives across the 7 datasets. Additionally tuning the kernel size in the first layer gives an accuracy that is better than 82% of alternative hyperparameter combinations, on average across all the datasets. It is worth noting also that increasing the number of filters in the first two layers to 32 gives a further boost so that the accuracy is higher than about 88% of alternative hyperparameter combinations. The number of filters in the first two layers seems to have increased importance in the specific context of a network tuned according to the advice we give above. In this context, the accuracy is better when using 32 and 32 instead of 16 and 8 filters in the first and second layers, providing another option for users to consider if they need to boost accuracy. Overall, putting our advice into practice appears to give competitive accuracy scores across multiple datasets.

6. Conclusion

In our introduction, we argued that while there are many successful neural network architectures for bearing fault detection, achieving good performance in real-world applications depends on how well the hyperparameters are tuned for a specific dataset. Benchmark datasets are useful for comparing algorithms but they might not reflect the real-world conditions where the algorithm will be applied, leading to a situation where an algorithm performs well on a benchmark dataset but poorly on a new dataset. Even small changes to hyperparameters can significantly impact the network's performance and yet there is often no clear guidance on how to choose the best hyperparameters in order to maximise accuracy on new data. We went as far as to say that explaining how hyperparameters affect accuracy on different datasets is more important than focusing on finding ever-newer neural network architectures and we overall argued for a shift in focus from developing new neural network architectures to explaining how to optimise existing ones for real-world applications through better hyperparameter tuning.

Past work showed that the wide-kernel architecture we focus on in this paper is sensitive to changes in the hyperparameters. Our new explorations of a transformer-based architecture and an LSTM architecture for fault detection showed that different kinds of neural network are greatly impacted by the values of the hyperparameters, with the accuracy dropping dramatically when the hyperparameters are set poorly. This serves to emphasise the value of knowing how to set the hyperparameters soundly.

Shifting our focus back to the wide-kernel CNN architecture, a lot of our research focused on the width of the kernel in the first layer. The kernel size is an important hyperparameter and a 'wide' kernel in particular has been pointed out in past literature as especially useful for processing raw sensor data. Making the first layer kernel too small causes a drop to 25% average accuracy on CWRU data from 69% when the kernel is large. We had expected the value of a wide kernel to be related to either the sampling rate of the data or the presence of high-frequency noise and yet our experiments showed that neither the sampling rate nor noise is responsible for wide-kernel CNNs performing better than narrow kernel networks. Instead, we are left speculating about what really causes wide kernels to perform better on CWRU data. One possibility is that the wide kernel provides some sort of generic architectural advantage such as regularisation that speeds up training and makes it easier for the training process to converge to a good solution. However, if the advantage of a wide kernel is in some way generic then it is not clear why the phenomenon fails to generalise to SEU data, where a larger kernel performs worse. There could be data properties driving the choice of kernel size which we are simply unaware of.

After considerable exploration our investigations led to some concrete advice on how to tune the hyperparameters: most of them can be set low in order to save computation, whilst the preferred number of filters in layer 1 and in layers 3-5 seems to be 16 or 32. The kernel size in the first layer is the most sensitive to differences between datasets, and it needs to be tuned carefully. Correctly tuned hyperparameters give the final touch to a wide-kernel CNN that performs well on multiple datasets yet also is small and lightweight, making it easier to train and potentially deploy in embedded systems. If performance is lower than expected when generalising to new data then the kernel size in the first layer and the number of filters in layers 3-5 have the highest feature importance and should be tuned first. If the network is still poor-performing on new data then we also suggest an order for tuning the most important hyperparameters individually (*Kernel size layer 1*, *Filters layer 1* and then *Filters layers 3-5*). Tuning the hyperparameters one-at-a-time saves a huge amount of computation compared to a full grid search and could be more realistic for practitioners to do.

We of course acknowledge there are limitations to our work. Limitations include the fact that we have to restrict our advice only to people analysing raw vibration signals sampled at a fairly high sampling rate. Our insights do not apply to other data, such as data unrelated to (potentially faulty) rotating machines. In other words, our insights are domain-specific. Another disclaimer that should be attached to our work is that although we tried to incorporate as much data as possible, there are only a few benchmark datasets in existence. Even when using seven datasets, we do not have a statistically robust sample, so our conclusions could be influenced by sampling error. It is important to check the results on new data. The accuracy scores reached by neural networks are strongly influenced by how they were trained and so it is also important to contemplate the details of our training approach. One consideration is that we restricted the number of epochs to 100 when training networks and applied early stopping if a network did not improve in validation loss for more than 10 successive epochs. The training procedure must end at some point and so it is necessary to have rules for when to stop in this manner. However, a possible consequence of our stopping rules is that we do not know if networks would have improved further had they been given more time to train. There is a chance that some networks which seem to perform poorly are actually effective but just relatively slow to train. At the same time, there is no reason to assume this is true and the stopping rules might have had no impact on the results. The limitation is that we cannot be completely certain.

Turning to look back at our research as a whole, we see many smaller analyses combining to give a larger picture of the hyperparameters for one specific wide-kernel CNN architecture and also how to set them. We see the hyperparameters impacting how accurate networks are on seven different benchmark datasets. Some hyperparameters can be kept at the same value when shifting between datasets, but the kernel size of the first layer (the so-called 'wide kernel' itself) needs to change in response to dataset properties. Multiple experiments with resampling and filtering sought but were ultimately unable to find the data property driving the correct choice of kernel size. Sitting next to these pieces of analysis, we also included an exploration of how the most important hyperparameters interact and which order to tune them in. Framing the entire analysis is our discussion of the importance of understanding hyperparameters when applying neural networks for bearing fault classification. Our main focus was to explain how to set the hyperparameters of a wide-kernel CNN architecture, and we concisely summarised the guidance we can give in that regard, following an extensive data-driven exploration.

CRediT Authorship Contribution Statement

Dan Hudson: Conceptualization, Investigation, Methodology, Project administration, Resources, Software, Validation, Visualization, Writing – original draft, Writing – review and editing
Jurgen van den Hoogen: Conceptualization, Data curation, Investigation, Methodology, Project administration, Resources, Software, Validation, Visualization, Writing – original draft, Writing – review and editing
Martin Atzmueller: Conceptualization, Methodology, Project administration, Supervision, Validation, Writing – review and editing

Funding

This work was supported by funds of zukunft.niedersachsen, Volkswagen Foundation (project "HybrInt – Hybrid Intelligence through Interpretable Artificial Intelligence in Machine Perception and Interaction").

Declaration of Competing Interest

The authors declare they have no financial and personal relationships with other people or organizations to disclose that could inappropriately influence or bias their work.

Data Availability

The data used in this research (results from the hyperparameter searches) are available on request.

References

[1] D. Hudson, J. van den Hoogen, S. Bloemheuvel, M. Atzmueller, Stay tuned! analysing hyperparameters of a wide-kernel architecture for industrial faults, in: 2024 IEEE Conference on Artificial Intelligence (CAI), 2024, pp. 1350–1356.

[2] J. van den Hoogen, D. Hudson, S. Bloemheuvel, M. Atzmueller, Hyperparameter analysis of wide-kernel cnn architectures in industrial fault detection – an exploratory study, Int. J. Data Sci. Anal. (2023). `doi:10.1007/s41060-023-00440-6`.

[3] W. Wahlster, From industry 1.0 to industry 4.0: towards the 4th industrial revolution (forum business meets research), Proceedings of the 3rd European Summit on Future Internet Towards. Future Internet International Collaborations. Espoo, Finland: Tivit (2012).

[4] H. Kagermann, W. Wahlster, Ten years of industrie 4.0, Sci 4 (3) (2022) 26.

[5] T. Zonta, C. A. Da Costa, R. da Rosa Righi, M. J. de Lima, E. S. da Trindade, G. P. Li, Predictive maintenance in the industry 4.0: A systematic literature review, Computers & Industrial Engineering 150 (2020) 106889.

[6] S. Vollert, M. Atzmueller, A. Theissler, Interpretable Machine Learning: A Brief Survey From the Predictive Maintenance Perspective, in: Proc. IEEE International Conference on Emerging Technologies and Factory Automation (ETFA 2021), IEEE, 2021, pp. 01–08.

[7] D. Neupane, M. R. Bouadjenek, R. Dazeley, S. Aryal, Data-driven machinery fault detection: A comprehensive review, arXiv preprint arXiv:2405.18843 (2024).

[8] W. Zhang, C. Li, G. Peng, Y. Chen, Z. Zhang, A deep convolutional neural network with new training methods for bearing fault diagnosis under noisy environment and different working load, Mechanical systems and signal processing 100 (2018) 439–453.

[9] L. Eren, Bearing fault detection by one-dimensional convolutional neural networks, Mathematical Problems in Engineering 2017 (1) (2017) 8617315.

[10] X. Guo, L. Chen, C. Shen, Hierarchical adaptive deep convolution neural network and its application to bearing fault diagnosis, Measurement 93 (2016) 490–502.

[11] T. Han, R. Ma, J. Zheng, Combination bidirectional long short-term memory and capsule network for rotating machinery fault diagnosis, Measurement 176 (2021) 109208.

[12] J. van den Hoogen, S. Bloemheuvel, M. Atzmueller, An improved wide-kernel cnn for classifying multivariate signals in fault diagnosis, in: 2020 International Conference on Data Mining Workshops (ICDMW), 2020, pp. 275–283. `doi:10.1109/ICDMW51313.2020.00046`.

[13] J. van den Hoogen, S. Bloemheuvel, M. Atzmueller, Classifying multivariate signals in rolling bearing fault detection using adaptive wide-kernel cnns, Applied Sciences 11 (23) (2021). `doi:10.3390/app112311429`.
URL `https://www.mdpi.com/2076-3417/11/23/11429`

[14] D. W. Kim, E. S. Lee, W. K. Jang, B. H. Kim, Y. H. Seo, Effect of data preprocessing methods and hyperparameters on accuracy of ball bearing fault detection based on deep learning, Advances in Mechanical Engineering 14 (2) (2022) 16878132221078494.

[15] W. Zhang, G. Peng, C. Li, Y. Chen, Z. Zhang, A new deep learning model for fault diagnosis with good anti-noise and domain adaptation ability on raw vibration signals, Sensors 17 (2) (2017) 425.

[16] S. J. Lacey, An Overview of Bearing Vibration Analysis, Brochure (2008).
URL `https://www.schaeffler.com/remotemedien/media/_shared_media/08_media_library/01_publications/schaeffler_2/technicalpaper_1/download_1/vibration_analysis_en_en.pdf`

[17] V. Pandhare, J. Singh, J. Lee, Convolutional neural network based rolling-element bearing fault diagnosis for naturally occurring and progressing defects using time-frequency domain features, in: 2019 Prognostics and System Health Management Conference (PHM-Paris), 2019, pp. 320–326. `doi:10.1109/PHM-Paris.2019.00061`.

[18] W. Saeed, C. Omlin, Explainable ai (xai): A systematic meta-survey of current challenges and future opportunities, Knowledge-Based Systems 263 (2023) 110273. `doi:https://doi.org/10.1016/j.knosys.2023.110273`.

URL https://www.sciencedirect.com/science/article/pii/S0950705123000230

[19] M. R. W. Group, et al., Report of large motor reliability survey of industrial and commercial installations, part i, IEEE Trans. Ind Appl. 1 (4) (1985) 865–872.

[20] S. Yin, X. Li, H. Gao, O. Kaynak, Data-based techniques focused on modern industry: An overview, IEEE Trans. Ind. Electron. 62 (1) (2014) 657–667.

[21] R. Zhao, R. Yan, Z. Chen, K. Mao, P. Wang, R. X. Gao, Deep learning and its applications to machine health monitoring, Mechanical Systems and Signal Processing 115 (2019) 213–237.

[22] D. Pandya, S. Upadhyay, S. P. Harsha, Fault diagnosis of rolling element bearing with intrinsic mode function of acoustic emission data using apf-knn, Expert Systems with Applications 40 (10) (2013) 4137–4145.

[23] Z. Zhou, C. Wen, C. Yang, Fault detection using random projections and k-nearest neighbor rule for semiconductor manufacturing processes, IEEE Transactions on Semiconductor Manufacturing 28 (1) (2015) 70–79. doi:10.1109/TSM.2014.2374339.

[24] Z. Wang, Q. Zhang, J. Xiong, M. Xiao, G. Sun, J. He, Fault diagnosis of a rolling bearing using wavelet packet denoising and random forests, IEEE Sensors Journal 17 (17) (2017) 5581–5588.

[25] P. Santos, L. F. Villa, A. Reñones, A. Bustillo, J. Maudes, An svm-based solution for fault detection in wind turbines, Sensors 15 (3) (2015) 5627–5648.

[26] D. You, X. Gao, S. Katayama, Wpd-pca-based laser welding process monitoring and defects diagnosis by using fnn and svm, IEEE Transactions on Industrial Electronics 62 (1) (2014) 628–636.

[27] J. Huang, X. Hu, F. Yang, Support vector machine with genetic algorithm for machinery fault diagnosis of high voltage circuit breaker, Measurement 44 (6) (2011) 1018–1027.

[28] P. Konar, P. Chattopadhyay, Bearing fault detection of induction motor using wavelet and support vector machines (svms), Applied Soft Computing 11 (6) (2011) 4203–4211.

[29] P. Malhotra, A. Ramakrishnan, G. Anand, L. Vig, P. Agarwal, G. Shroff, Lstm-based encoder-decoder for multi-sensor anomaly detection, arXiv preprint arXiv:1607.00148 (2016).

[30] P. Yao, S. Yang, P. Li, Fault diagnosis based on rsenet-lstm for industrial process, in: 2021 IEEE 5th Advanced Information Technology, Electronic and Automation Control Conference (IAEAC), Vol. 5, 2021, pp. 728–732. `doi:10.1109/IAEAC50856.2021.9391030`.

[31] T. Ince, S. Kiranyaz, L. Eren, M. Askar, M. Gabbouj, Real-time motor fault detection by 1-d convolutional neural networks, IEEE Trans. Ind. Electron. 63 (11) (2016) 7067–7075.

[32] W. Zhang, G. Peng, C. Li, Rolling element bearings fault intelligent diagnosis based on convolutional neural networks using raw sensing signal, in: Advances in Intelligent Information Hiding and Multimedia Signal Processing, Springer, 2017, pp. 77–84.

[33] J. van den Hoogen, S. Bloemheuvel, M. Atzmueller, An improved wide-kernel cnn for classifying multivariate signals in fault diagnosis, in: 2020 International Conference on Data Mining Workshops (ICDMW), 2020, pp. 275–283. `doi:10.1109/ICDMW51313.2020.00046`.

[34] J. van den Hoogen, S. Bloemheuvel, M. Atzmueller, Classifying multivariate signals in rolling bearing fault detection using adaptive wide-kernel cnns, Applied Sciences 11 (23) (2021). `doi:10.3390/app112311429`.
URL `https://www.mdpi.com/2076-3417/11/23/11429`

[35] F. Rosenblatt, The perceptron: a probabilistic model for information storage and organization in the brain., Psychological review 65 (6) (1958) 386.

[36] A. Krizhevsky, I. Sutskever, G. E. Hinton, Imagenet classification with deep convolutional neural networks, in: Advances in neural information processing systems, 2012, pp. 1097–1105.

[37] K. Simonyan, A. Zisserman, Very deep convolutional networks for large-scale image recognition, arXiv preprint arXiv:1409.1556 (2014).

[38] Y. LeCun, Y. Bengio, et al., Convolutional networks for images, speech, and time series, The handbook of brain theory and neural networks 3361 (10) (1995) 1995.

[39] A. Hannun, C. Case, J. Casper, B. Catanzaro, G. Diamos, E. Elsen, R. Prenger, S. Satheesh, S. Sengupta, A. Coates, et al., Deep speech: Scaling up end-to-end speech recognition, arXiv preprint arXiv:1412.5567 (2014).

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

[41] M. P. Naeini, H. Taremian, H. B. Hashemi, Stock market value prediction using neural networks, in: 2010 international conference on computer information systems and industrial management applications (CISIM), IEEE, 2010, pp. 132–136.

[42] K. Abhishek, M. Singh, S. Ghosh, A. Anand, Weather forecasting model using artificial neural network, Procedia Technology 4 (2012) 311–318.

[43] A. Hajnayeb, A. Ghasemloonia, S. Khadem, M. Moradi, Application and comparison of an ann-based feature selection method and the genetic algorithm in gearbox fault diagnosis, Expert systems with Applications 38 (8) (2011) 10205–10209.

[44] Y. Liu, X. Yan, C.-a. Zhang, W. Liu, An ensemble convolutional neural networks for bearing fault diagnosis using multi-sensor data, Sensors 19 (23) (2019). `doi:10.3390/s19235300`.
URL `https://www.mdpi.com/1424-8220/19/23/5300`

[45] J. Yang, M. N. Nguyen, P. P. San, X. Li, S. Krishnaswamy, Deep convolutional neural networks on multichannel time series for human activity recognition., in: Ijcai, Vol. 15, Buenos Aires, Argentina, 2015, pp. 3995–4001.

[46] Y. Zheng, Q. Liu, E. Chen, Y. Ge, J. L. Zhao, Time series classification using multi-channels deep convolutional neural networks, in: International Conference on Web-Age Information Management, Springer, 2014, pp. 298–310.

[47] A. Zhang, S. Li, Y. Cui, W. Yang, R. Dong, J. Hu, Limited data rolling bearing fault diagnosis with few-shot learning, IEEE Access 7 (2019) 110895–110904.

[48] D. Zhao, T. Wang, F. Chu, Deep convolutional neural network based planet bearing fault classification, Computers in Industry 107 (2019) 59–66.

[49] R. Chen, X. Huang, L. Yang, X. Xu, X. Zhang, Y. Zhang, Intelligent fault diagnosis method of planetary gearboxes based on convolution neural network and discrete wavelet transform, Computers in Industry 106 (2019) 48–59.

[50] Z. Zhao, Y. Jiao, X. Zhang, A fault diagnosis method of rotor system based on parallel convolutional neural network architecture with attention mechanism, Journal of Signal Processing Systems (2023) 1–13.

[51] R. Li, J. Wu, Y. Li, Y. Cheng, Periodnet: Noise-robust fault diagnosis method under varying speed conditions, IEEE Transactions on Neural Networks and Learning Systems (2023) 1–15doi:10.1109/TNNLS.2023.3274290.

[52] S. Chen, J. Yu, S. Wang, One-dimensional convolutional auto-encoder-based feature learning for fault diagnosis of multi-variate processes, Journal of Process Control 87 (2020) 54–67. doi:https://doi.org/10.1016/j.jprocont.2020.01.004.
URL https://www.sciencedirect.com/science/article/pii/S0959152419300708

[53] Y. LeCun, L. Bottou, Y. Bengio, P. Haffner, Gradient-based learning applied to document recognition, Proceedings of the IEEE 86 (11) (1998) 2278–2324.

[54] I. Goodfellow, Y. Bengio, A. Courville, Deep learning, MIT press, 2016.

[55] C. Gulcehre, M. Moczulski, M. Denil, Y. Bengio, Noisy activation functions, in: International conference on machine learning, PMLR, 2016, pp. 3059–3068.

[56] S. Kiranyaz, T. Ince, R. Hamila, M. Gabbouj, Convolutional neural networks for patient-specific ecg classification, in: Proc. IEEE EMBC, IEEE, 2015, pp. 2608–2611.

[57] A. F. Agarap, Deep learning using rectified linear units (relu) (2019). `arXiv:1803.08375`.
URL `https://arxiv.org/abs/1803.08375`

[58] I. Priyadarshini, C. Cotton, A novel lstm–cnn–grid search-based deep neural network for sentiment analysis, The Journal of Supercomputing 77 (12) (2021) 13911–13932.

[59] D. Jana, J. Patil, S. Herkal, S. Nagarajaiah, L. Duenas-Osorio, Cnn and convolutional autoencoder (cae) based real-time sensor fault detection, localization, and correction, Mechanical Systems and Signal Processing 169 (2022) 108723.

[60] S. Gupta, Deep learning based human activity recognition (har) using wearable sensor data, International Journal of Information Management Data Insights 1 (2) (2021) 100046.

[61] G. Kalouris, E. I. Zacharaki, V. Megalooikonomou, Improving cnn-based activity recognition by data augmentation and transfer learning, in: 2019 IEEE 17th International Conference on Industrial Informatics (INDIN), Vol. 1, IEEE, 2019, pp. 1387–1394.

[62] H. Yi, K.-H. N. Bui, An automated hyperparameter search-based deep learning model for highway traffic prediction, IEEE Transactions on Intelligent Transportation Systems 22 (9) (2020) 5486–5495.

[63] S. S. Mostafa, F. Mendonca, A. G. Ravelo-Garcia, G. G. Juliá-Serdá, F. Morgado-Dias, Multi-objective hyperparameter optimization of convolutional neural network for obstructive sleep apnea detection, IEEE Access 8 (2020) 129586–129599.

[64] R. Zatarain Cabada, H. Rodriguez Rangel, M. L. Barron Estrada, H. M. Cardenas Lopez, Hyperparameter optimization in cnn for learning-centered emotion recognition for intelligent tutoring systems, Soft Computing 24 (10) (2020) 7593–7602.

[65] X. Zhang, X. Chen, L. Yao, C. Ge, M. Dong, Deep neural network hyperparameter optimization with orthogonal array tuning, in: Neural Information Processing: 26th International Conference, ICONIP 2019, Sydney, NSW, Australia, December 12–15, 2019, Proceedings, Part IV 26, Springer, 2019, pp. 287–295.

[66] Cwru dataset; case western reserve university bearing data center, available: https://csegroups.case.edu/ bearingdatacenter/home.

[67] D. Neupane, J. Seok, Bearing fault detection and diagnosis using case western reserve university dataset with deep learning approaches: A review, IEEE Access 8 (2020) 93155–93178.

[68] X. Liu, H. Huang, J. Xiang, A personalized diagnosis method to detect faults in a bearing based on acceleration sensors and an fem simulation driving support vector machine, Sensors 20 (2) (2020) 420.

[69] F. Piltan, J.-M. Kim, Svm-based hybrid robust pio fault diagnosis for bearing, in: International Conference on Intelligent and Fuzzy Systems, Springer, 2020, pp. 858–866.

[70] C. Lessmeier, J. K. Kimotho, D. Zimmer, W. Sextro, Condition monitoring of bearing damage in electromechanical drive systems by using motor current signals of electric motors: A benchmark data set for data-driven classification, Proc. PHM Society European Conference 3 (1) (2016).

[71] H. Malik, Y. Pandya, A. Parashar, R. Sharma, Feature extraction using EMD and classifier through artificial neural networks for gearbox fault diagnosis, in: Applications of Artificial Intelligence Techniques in Engineering: SIGMA 2018, Volume 2, Springer, 2019, pp. 309–317.

[72] Y. Pandya, Gearbox fault diagnosis data (06 2018).
URL `https://data.openei.org/submissions/623`

[73] Society For Machinery Failure Prevention Technology, Fault Data Sets, `https://mfpt.org/fault-data-sets/`, accessed: July 2023 (Online).

[74] B. Wang, Y. Lei, N. Li, N. Li, A hybrid prognostics approach for estimating remaining useful life of rolling element bearings, IEEE Transactions on Reliability (2018) 1–12`doi:10.1109/TR.2018.2882682`.

[75] P. Cao, S. Zhang, J. Tang, Gear Fault Data (4 2018). `doi:10.6084/m9.figshare.6127874.v1`.
URL `https://figshare.com/articles/dataset/Gear_Fault_Data/6127874`

[76] P. Cao, S. Zhang, J. Tang, Preprocessing-free gear fault diagnosis using small datasets with deep convolutional neural network-based transfer learning, Ieee Access 6 (2018) 26241–26253.

[77] S. U. (SEU), Gearbox mechanical datasets, `https://github.com/cathysiyu/Mechanical-datasets`, accessed: July 2023 (Online).

[78] X. Chen, B. Zhang, D. Gao, Bearing fault diagnosis base on multi-scale cnn and lstm model, Journal of Intelligent Manufacturing 32 (4) (2021) 971–987.

[79] S. Hochreiter, Long short-term memory, Neural Computation MIT-Press (1997).

[80] A. Vaswani, Attention is all you need, Advances in Neural Information Processing Systems (2017).

[81] R. Qiang, X. Zhao, An intelligent diagnosis method for rolling bearings based on ghost module and adaptive weighting module, Research Square Preprint https://doi.org/10.21203/rs.3.rs-2627489/v1 (2023).

[82] A. Zhou, A. B. Farimani, Faultformer: Pretraining transformers for adaptable bearing fault classification, IEEE Access (2024).

[83] D. P. Kingma, J. Ba, Adam: A method for stochastic optimization, arXiv preprint arXiv:1412.6980 (2014).

[84] L. S. Shapley, A value for n-person games, Contribution to the Theory of Games 2 (1953).

[85] A. Brandsæter, I. K. Glad, Shapley values for cluster importance, Data Mining and Knowledge Discovery (2022) 1–32.

[86] J. Kline, Properties of the d-dimensional earth mover's problem, Discrete Appl. Math. 265 (2019) 128–141. `doi:https://doi.org/10.1016/j.dam.2019.02.042`.