Optimizing deep learning inference for resource-constricted platforms

Adela Nedisan Videsjorden

Thesis submitted for the degree of
Master in Informatics: Programming and System Architecture
60 credits

Institute for Informatics
Faculty of mathematics and natural sciences

UNIVERSITY OF OSLO

Spring 2023

Optimizing deep learning inference for resource-constricted platforms

Adela Nedisan Videsjorden

Optimizing deep learning inference for resource-constricted platforms

http://www.duo.uio.no/

Printed: Reprosentralen, University of Oslo

Abstract

The last decade has witnessed tremendous advances within the realm of industrial applications, at the core of which stands a recent fusion between sensing and artificial intelligence (AI) technologies. By design, deep learning models are particularly well-suited at capturing hidden patterns and correlations in masses of heterogeneous data, displaying exceptional performance compared to other traditional learning methods. Their powerful predictive ability is however eclipsed by their large computational requirements. Offloading segments or entire computational processes onto the cloud is not always feasible due to privacy concerns, limited bandwidth and minimal latency tolerance. To overcome these challenges, a promising strategy involves moving all computations near or at the edge level, reducing transmission costs. Contrasted to cloud servers, edge devices operate under constrained conditions, and are characterized by reduced memory, power and computational capacities. Accommodating deep learning models on resource-constricted devices thus translates to finding those optimization strategies that best balance the trade-off between predictive power and resource utilization. Existing implementations concentrate on increasing the model accuracy, and often neglect reducing resource footprint. Although many edge intelligence-enabling hardware and software optimizations have been proposed, they are often tailored to specific architectures and contexts. This thesis proposes a framework for finding a one-fits-all solution for efficient edge deployment and inference, by exploring different compression techniques across various model architectures and complexities. The design and implementation choices we make are based on a literature review on latest optimization trends. This serves as a guide for research aimed at fitting trained models onto resource-constricted devices. A critical ingredient to our approach is the comparative analysis between full-size and optimized model counterparts, that allows us to better capture the relationship between predictive accuracy and resource utilization. Among many possible use cases, we select a healthcare application scenario where we deploy models built for stress detection, a non-linear time-series analysis problem, onto an edge device.

Acknowledgements

This thesis has materialized thanks to those who have contributed not only by knowledge sharing but also by words of encouragement throughout these past few months.

I would like to express my gratitude towards my thesis supervisors Rustem Dautov and Erik Johannes Husom for their continuous support, invaluable advice and feedback, whose expertise has not only led to the completion of this thesis, but also to strengthening my writing and experimenting skills. A heartful “thank you” goes to my internal supervisor Ketil Stølen for believing in my capabilities and for putting me in contact with the right people at the right time, his input leading to many positive developments. I also would like to thank my colleagues at SINTEF for always lifting up my spirit and putting me on the right track with my setup.

Last but not least, a lot of love and appreciation goes to my family and friends: my mom Ileana and my grandparents for always keeping me in their prayers, my dad Ioan for his moral support, my brother Radu for being my source of inspiration, my in-laws and my friends for being the best cheerleaders. A special thank you goes to my husband Halvor for always believing in me and for being my stone throughout the whole process.

Contents

List of Tables

List of Figures

1 Introduction

In the light of the fourth industrial revolution, more and more industry sectors undergo a gradual transition to a greater digitally-empowered workforce, as innovative technologies emerge providing quicker and more reliable solutions to workplace-related problems. Such novel technological developments often register the application of AI, particularly Machine Learning (ML) models that can be employed in order to make fast, ideally real-time, high-grade predictions based on behavioral and physiological data collected within the work environment. Generating a prediction or an output is also commonly known as inference and typically entails having access to a pre-trained model. Nevertheless, the inference process may not only be computationally intensive, but also use significant amounts of resources in terms of energy, power and memory footprint. This is all the more true for models of high-complexity such as neural networks where time and space complexity is magnified by input dimensions and model architecture choice. Cloud servers can deal with computation-heavy processes, but cloud computing will sometimes be unfeasible and disadvantageous due to various factors, such as network outages, increased server or network costs and risks related to privacy. On top of this, data transmission remains constrained by the limitations of the network bandwidth, making cloud computing more prone to training bottlenecks and inference latency, thus less compatible with processing time-sensitive data.

On-device deployment and inference of ML models offer a potential workaround to the aforementioned issues. Edge devices are however prohibitive due to memory, power and computational requirements. The solution to this issue constitutes in resource optimization strategies applied at the hardware and at the software level respectively. Nevertheless, these too carry an associated cost, which translates to a degree of accuracy loss. The challenge at hand consists in balancing the trade-off between edge resource utilization and model performance preservation. While current research efforts are focused on optimization strategies

for specific scenarios, there is a clear lack of research aimed towards finding a widely-applicable optimization strategy that covers not only several model architectures, but also adds a complexity dimension.

The purpose of this study will be to give an answer to the question of *whether it is possible to find a generalized set of optimization techniques that allows efficient model deployment and inference on resource-constricted devices*. To this intent, we design and implement a Proof of Concept (PoC) framework by taking architectural, hardware- and evaluation-related decisions grounded in a comprehensive investigation on state-of-the-art techniques. This framework helps us acquire an one-fits-all solution to the issues mentioned previously. Our area of interest is centered exclusively around model optimizations and design choices that lead to so-called "lightweight" architectures, as they appear in current research. Drawing upon these insights, we propose a framework that allows us to find whether a subset of these optimization techniques can be generally applied across model architectures, highlighting those that are most successful in balancing out the trade-off between performance preservation and resource utilization.

We increase robustness by running two separate experiments, after generating a large set of learning models belonging to a wide range of architectures and displaying multiple degrees of complexity. The full-sized models are then subjected to various optimization techniques, "lightweight" versions emerging as byproducts of this process. To this pipeline we then add deploying these models to a widely-used edge device and simulating inference on it. A shared set of key performance indicators is then used to extract relevant inference evaluation metrics for both the full-sized models and their lightweight counterparts. This evaluation scheme lies at the very core of our PoC and enables us to set up a tight comparative analysis that considers all specified dimensions. Our search for widely-applicable optimizations prompts an extensive discussion at the end of the thesis, where we validate our results by defining a set of success criteria.

In this whole process one crucial step is fitting our models, but to do so we must first find an appropriate data set. Time-series data processing will be prioritized in this thesis not only because it allows us to cover a broader range of model architectures, but it is also found at the base of many valuable real-time industrial applications. To this end, we will use the WESAD data set which provides quantitative variables acquired by a suite of wearable devices, with multiple applications belonging to affective computing. In this data set the physiological characteristics of an individual at any given time can be used as an indicator of their psychological state, whether it is neutral, amused, or stressed. With the thesis objective in mind, our efforts will be directed towards preserving the initial performance of the full-sized models after optimizations are applied.

In the subsequent sections, we will follow the above steps implementing a PoC that aims to fulfill all defined success criteria, allowing us to answer the posed research question and fill in multiple research gaps. This means achieving a better understanding of how the envisioned framework is built, how it allows us to discover whether ML optimization can be applied across several dimensions, but also which techniques stand out the most in this respect.

1.1 Thesis Outline

The thesis structure reflects the evolving nature of the thesis starting from planning, going through the implementation and ending with evaluation. Section 2 pieces together an all-encompassing thesis statement in subsection 2.1, which is a clear definition as to what our PoC is expected to be and do. In subsection 2.2 the thesis statement is later characterized by two success criteria. Next we have section 3 which covers the necessary background. Subsection 3.1 provides a deeper understanding of thesis-relevant AI/ML concepts. This is followed by subsection 3.2, which illustrates the usage of AI/ML-fused technology across various industries. Subsection 3.3 offers a description of time-series data. Section 4 is concerned with planning and performing the literature review. Subsection 4.1 initializes the process by describing a literature review protocol showcasing the steps needed to perform a systematic literature review. Here we present some initial results as we follow the protocol actions. A fuller report on findings can be found in subsection 4.2, where we extract relevant knowledge about current trends and state-of-the-art architectures, edge hardware, evaluation strategies and resource-oriented optimizations. This is followed by subsection 4.3 where research gaps are highlighted, and lastly subsection 4.4 where our contributions are listed. Section 5 showcases the developmental stages and documents these. Subsection 5.1 goes over the data set particularities, the model generation and training phases, while subsection 5.2 explains how these models are optimized. The next subsection is 5.3, which goes over the details of the deployment and inference phases. Next to last we have section 6 where we present the experimental results of full-sized models and the comparative analysis between full-size models and their optimized counterparts. Finally, section 7 provides a discussion on findings linked to our PoC. We follow this by a take on faced limitations of our approach, describing also how our efforts helps pave a path for future work.

2 Thesis Statement and Success Criteria

2.1 Thesis Statement

Before equipping ourselves with a deeper understanding about bettering deployment and inference on edge devices, we should focus on the type of framework that we envision being built. We base our PoC description on an intuition around its qualities and desired functionality, then we encompass these thoughts into a simple thesis statement. As mentioned in the introduction, by the end of the thesis we should be able to find a rather generalized set of optimization techniques which can to a certain degree be reused under different scenarios with the goal of reducing the load on resources with minimal performance loss. In more technical terms, this set could be ascribed to a tool which outputs an efficiently optimized version of an input model, regardless of its original architecture and complexity. Finding a fair balance between the available computing resources and the performance of the predictive models can be regarded as the most vital component of this tool, and thus we regard this optimization set as sufficient proof of its existence. Let us now formulate the problem statement in more general terms. The statement constitutes the basis on which we build our success criteria in the next subsection, so it should be short and concise. We will thus make the following statement, and claim that:

It is possible to develop a practical ML-based approach for time-series data analysis, in such a way that the model can be deployed and executed on resource-constricted devices.

We want to support this statement by implementing and evaluating the PoC. This involves following a series of steps that take consideration of current trends and technology. Being able to support this statement and finding ways of fulfilling it is however not possible without understanding all of the terms that are embedded in the statement. We take a closer look at our thesis statement and try to solve some term ambiguities in order to get a complete understanding of what our final goal is.

In the statement, the term *practical* can be interpreted in several ways. So what exactly do we mean by practicality in this context? In subsequent sections we will see that our design entails generating full-sized models and then applying optimizing techniques post-training, resulting in new, optimized models. A practical solution in this context refers to the ability of preserving resources on the targeted environment and minimizing model performance loss once the model is deployed and executed. To be more precise, we argue that a practical model should be able to preserve most if not all of its accuracy once optimized, and use minimal device resources by keeping a low footprint. The next term we consider is "approach". *Approach* should not be interchanged with "optimization set", because while we are indeed employing the use of various optimizations, an approach carries much more meaning and includes the investigation of various techniques and all decision-making prior to building the framework. *Time-series data* is merely a mention of the type of data we wish to process, and its choice will be further justified in coming sections. The *model* term here stands for any architecture of neural network from those that we will cover. Lastly, we highlight that the approach should return models that are *deployable* and *executable*. These two terms connect to our previous description of "practical". We will see how everything translates to the success criteria.

2.2 Success Criteria

Success criteria point to the sort of attributes that we want to have present in our optimized models, and provide a method of validating whether our approach has been successful. Checking against these criteria determines whether our framework does what it is expected to, and if our thesis goal has been met. These connect to our previous statement, as we are going to define what a "practical" solution would be in terms of model qualities and requirements. We can now describe these success criteria, as follows:

1. **Accuracy Success Criteria:** The optimized model should provide predictions with an accuracy comparable to that of its full-size model.

2. **Resource Success Criteria:** The optimized model should minimize resource utilization so that it is below that of the full-size model.

For the first criteria, the optimized model accuracy is comparable to the full-size counterpart if it is equal, higher, or if it reduced by a small predetermined value given in terms of accuracy loss points. We choose a threshold of 0.1, and claim that anything above this accuracy drop should not be permitted, as models need to have a similar accuracy to that of their full-sized counterparts. To fulfill this requirement, this has to be true across all of the model architectures, and across all of the complexities (i.e model sizes). For the second criteria, the performance of the optimized model here is related to its resource utilization. This is a requirement to minimize its impact on device resources. In quantitative terms, the optimized model should optimally use less resources than its full-sized counterpart in order to be deemed practical. Which resources are deemed relevant for various constricted devices is one of topics to be discussed in later sections. If there are several optimizations that fulfill this requirement, then the ones that uses least resources are preferred. Note that while these two are listed separately, the trade-off between the two is also a dimension of interest, since we do not want to compromise too much on one for the other. Even if not listed explicitly as a success criteria, it should be possible to deploy the optimized model on devices where the full-size model fails to do so. This is imperative for highly resource-constricted devices.

All in all, we have several important success criteria which we want to fulfill and which we will have to take consideration of when implementing our PoC. Evaluating the PoC will be done in respect to these success criteria. Note that the actual evaluation is done on the optimized models, however finding these optimizations in the first place supports our thesis statement.

3 Research Context and Motivation

Before we dig deeper into the topic of resource-efficient optimization and the actual implementation and evaluation stages of our PoC, we first have to highlight the use and importance of AI/ML technologies in an industrial setting by discussing how deployment on resource-constricted devices has unlocked new types of applications, but also briefly outline some of the challenges that are met. We follow this up by a few elementary notions on time-series data processing. To familiarize the reader with some of the terminology occurring in later sections and establish a solid knowledge foundation on AI and ML, we begin this section with a short introduction around these concepts.

3.1 Artificial Intelligence and Machine Learning

Although we assume that the reader has some basic understanding on topics of this field, this section will go over key concepts and further introduce the *AI*, *ML*, *neural network* and *deep learning* terms, as they are featured frequently in later sections.

AI has been defined as a system's ability to interpret external data correctly and to achieve specific goals and tasks through flexible adaptation (Kaplan and Haenlein, 2019), a capability that can be embedded into machines, giving them learning abilities without needing to be explicitly programmed (Karthikeyan et al., 2019). This learning ability is usually the result of a process called *training*, which means fitting a model to a data set. Once trained, the model can be used to predict outcomes based on previously seen data, a process called *inference*.

ML is a subdomain of AI, which according to Marsland, 2011 encompasses a set of four learning paradigms: *supervised learning*, *unsupervised learning*, *reinforcement learning* and *evolutionary algorithms*. In this section and overall thesis we will consider only algorithms belonging to the first category. Supervised learning refers to methods that are trained using data where instances are associated with one or more labels. These labels can either belong to a discrete set of classes, or

take the form of numerical values. The first type of labels are used in classification problems, where we make a discrete prediction by mapping instances to one or several classes, normally after finding a good decision boundary that separates the different classes. The second type of labels are used in regression tasks where we wish to predict continuous numerical values, usually by fitting a shape as closely to the training data as possible. Supervised methods can be very effective at determining the label of any unseen data point, given that the sampled data set is well representative of the population and the model has been properly trained. This ability is called *generalization*. Models should be able to generalize well for new instances, meaning that they should capture the overall structure in the training data (otherwise the model is considered *underfit*) and avoid capturing its peculiarities (otherwise the model is considered *overfit*). Some algorithms are instance-based and need no prior training before outputting a prediction, going under the category of *lazy learning* algorithms (Aha, 2013). These differ from *eager learning* algorithms which have to be trained prior to making any predictions.

Another distinction should be made between traditional ML learning and *deep learning*. The first is characterized by the use of handcrafted features, or features selected by means of *feature engineering*, which are often a prerequisite to good performance. Deep learning algorithms decipher on their own which parts of the input weigh most in the correction of prediction outcomes. This section and thesis will feature only deep learning models. There are three main reasons for this. The first is that handcrafted features have a few drawbacks: they typically require domain knowledge and developers have to go through many selection trials before picking a good set. Secondly, many traditional ML models are also limited to achieving good performance only for linearly separable data sets, but such data sets are not as prevalent. Deep learning models are thus preferred due to their strong nonlinear representation learning capabilities (Zhong et al., 2016). Lastly, complex deep learning models are also more heavily parameterized, and so resource optimization is of higher interest for this category of mod-

els. Although there are cases where smaller, non-neural models outperform neural ones (Thompson et al., 2020), deep learning models usually show outstanding performance compared to traditional ML models in many different applications, for example sentiment analysis (Kansara and Sawant, 2020, Hassan and Mahmood, 2017), face recognition (Setiowati et al., 2017), disease detection (Sahu et al., 2020, Iqbal et al., 2021, Shruthi et al., 2019), to name a few. Another choice for our thesis will be the focus on inference. While the main reason for this is narrowing our study field, Intel predicted that by 2020 the ratio between training and inference will increase five times for inference alone, with inference taking up to 80% of the AI workflows (Intel, n.d.), suggesting that research directed towards improving inference is indeed of high importance. We also focus on classification only, primarily because, structurally speaking, the deep learning model architectures do not change much between regression and classification (the output layer registers a change in activation function). There is also a possibility of turning regression tasks into classification by setting problem-dependent thresholds for learned probabilities, a technique called thresholding (Google, 2022), however the previous consideration still applies. On the other hand, the number of output neurons changes and has a bigger impact on the overall size and complexity of the model when moving from a binary to a multi-class task.

The purpose of learning is to get better at predicting labels for unseen data points. In order to be able to learn, we have to know whether or not our predictions are close or equal to the ground truth, which for supervised tasks is done by comparing predicted outputs to the existent labels. We thus need a way to measure for how well the current trained model generalizes, so that we can decide whether or not the training has been sufficient. The most straightforward way is to measure the model accuracy or performance in terms of prediction successes. The confusion matrices shown in figure 1 can be a helpful tool in understanding how performance evaluation metrics are calculated for both the binary and the multi-class classification tasks. These will be used later when we define our performance indicators. Here TP (True Positive) and TN

a)

Confusion Matrix (Binary)		Actual	
		True	False
Predicted	True	True Positives (TP)	False Positives (FP)
	False	False Negatives (FN)	True Negatives (TN)

b)

Confusion Matrix (Multi-class)		Actual		
		Class A	Class B	Class C
Predicted	Class A	True (Class A)	False (AB)	False (AC)
	Class B	False (BA)	True (Class B)	False (BC)
	Class C	False (CA)	False (CB)	True (Class C)

Figure 1: Confusion matrices: a) binary, b) multi-class.

(True Negative) stands for the number of correct predictions of positive and negative class, respectively. FN (False Negative) and FP (False Positive) are the numbers of actual positive predictions missed by the model and false predictions to the positive class. For the multi-class case, we can consider all values on a row (except the true class) as FP values, while all values on a column (except the true class) are FN values for that specific class.

Armed with a better understanding of AI, ML and deep learning, we will present some common deep learning architectures. *Artificial Neural Networks* (ANNs) are considered eager learning algorithms. They have gained immense momentum in the past years, kicked off by the high growth and availability of data generated by ubiquitous devices (Pouyanfar et al., 2018). In this thesis we will consider two types of ANNs: *acyclical* or *feed-forward* ANNs, and *cyclical* or *recurrent* ANNs. In short, we can distinguish them by their connections: feed-forward network connections do not form cycles, while connections in recurrent networks do (Graves, 2012). Deep learning models can have different depths and widths, from networks of low complexity displaying very few layers and units, also called *shallow networks*, to very large ones called *deep networks*. There is a general agreement that large models with many parameters are better performance achievers (Thompson et al., 2020), feed-forward multi-layer networks even earning the name *universal function approximators* (Hornik et al., 1989). But overparam-

eterizing, if not properly regularized, might lead to networks that learn too much about about the particularities of the training data, and become overfit. This is observed in practice when the network performs well on the training set but much worse on the validation set. Conversely, if a model is too small or not trained long enough, it has higher chances of being underfit. We observe this in practice when the network has low performance on both the training and validation data sets. In addition to that, deep networks may also suffer from the *vanishing gradient problem* which causes the computed gradients to get smaller and smaller during training thus making the process very slow. On the other hand we also have the *exploding gradient problem* which causes weights to be become very large, thus making the network sensitive to changes in input. These two issues can even happen concomitantly, however there are a few strategies used to avoid these problems, which we will address in the developmental phase of our PoC.

For now we will consider three variants of ANNs: the feed-forward *Multi-Layer Perceptron* (MLP), the *Convolutional Neural Network* (CNN), and the *Long Short-Term Memory* (LSTM), the latter being a type of *Recurrent Neural Network* (RNN). Vanilla RNNs are often subjected to exploding and vanishing gradients, but also memory constraints as long-term dependencies are difficult to handle. The chain of gradient multiplication becomes too long to model long-range dependencies and causes the vanishing gradient (Pal and Prakash, 2017). This is why other variants of RNNs have been proposed, which prove to be more successful in mitigating these issues, such as *Gated Recurrent Units* (GRU) (Cho et al., 2014), *Echo State Networks* (ESN) (Jaeger, 2001), or LSTMs (Hochreiter and Schmidhuber, 1997). This section and thesis considers LSTMs only, as their popularity exceeds that of other recurrent architectures, but also due to the mentioned benefits. For more information about our selection, we recommend reading the respective chapters and subchapters in Goodfellow et al., 2016.

3.1.1 Multi-Layer Perceptron

The feed-forward MLP is a fully-connected ANN, composed of one input layer, one or more hidden layers and one output layer. The units in a MLP are called neurons or perceptrons. The input layer contains as many neurons as there are input variables, while the output layer contains as many neurons as the number of classes or labels. The number of neurons on the hidden layers can vary. All layers are fully connected, which means that each node from each layer (except for the input layer) has connections to all of the neurons on the previous layer, and each connection has a weight attributed to it. All computations take place in these neurons, first by applying a bias, then summing the weighted inputs from the previous layer, and finally transforming the result by applying an non-linear activation function whose output indicates whether that specific neuron has been activated or not. Activated neurons produce output signals which are equal to their contributions to the final output of the network. The network is trained end-to-end by back-propagation, updating weights by gradient descent in terms of minimizing a loss function, starting from the weights going from the last layer into the output layer and down to the weights connecting the input layer to the first layer of neurons. To predict, we feed the new inputs through a trained MLP, do a forward pass by computing activations in all neurons and observe which label is predicted by the output node(s). Thus, a MLP is specified by the number of hidden layers the number of layers, the number of nodes within each layer, and the non-linearities that are used. When it comes to efficiency in processing time-series data, we will see that MLP is a slightly weaker candidate. For more information on back-propagation, we recommend reading Nielsen, 2015.

3.1.2 Convolutional Neural Network

The CNN is a very popular type of ANN, typically employed in visual imagery tasks (e.g. image and video processing), but some variants are successfully used for processing time-series data as well. The typical CNN consists of at least one convolutional layer, followed by one or sev-

eral fully-connected layers. In the MLP layer every output unit interacts with every input unit, while in the convolutional layer, that is not necessarily true. This is due to sparse interactions: filters (or kernels), typically smaller than the input itself, are shifted over the input data and across input channels, outputting feature maps which contain the resulting dot products. Filter parameters are learned during training and allow us to capture important patterns that lead us to correct predictions. It is common practice to further reduce the size of the convolutional layer output by the use of pooling functions. What pooling does is summarize the values belonging to a neighborhood. This neighbourhood can either be a local or global, the former considering part of the input as neighbourhood and the latter considering the whole input. Apart from reducing the number of computations, pooling layers also help make the representation approximately invariant to small translations or shifts of the input (Goodfellow et al., 2016). The reduction criteria is empirically chosen, whether we want to keep the maximum values (i.e max pooling) or their average (i.e average pooling) depends a lot on the specific task and what kind of features we want to preserve. Lastly, we have the fully-connected layer(s), or the classification part of the architecture where the output from the last convolutional layer is first flattened and then used to make actual predictions. While implementing CNNs, we will see that the input shape determines the dimensions of the filters used in the convolutional layers. For time-series data we are mainly interested in learning one-dimensional filters.

3.1.3 Long Short-Term Memory

The LSTM is a sequence-specialized type of ANN architecture, well suited for capturing temporal phenomena such as in the case of time-series data and natural language. The LSTM is an optimized type of RNN. Although the inner workings are very similar to those of the MLP, the forward and backpropagation phases are slightly different. In the forward phase the hidden layers take input from both the current and previous timestep. This is possible because in this architecture the

hidden units have recurrent connections that allow outputs from previous states to be inserted into current state calculations. This creates a form for memory. Because of this, learning through backpropagation (here called *backpropagation through time*) for recurrent networks also considers the current weight influence on the hidden layer at the next timestep. Initially proposed in 1997 (Hochreiter and Schmidhuber, 1997), the LSTM mitigates the issues that follow with vanishing gradient problems in vanilla RNNs. In addition to the outer recurrence of the RNN, LSTM networks contain "LSTM cells" with internal recurrence (self-loop) (Goodfellow et al., 2016). The LSTM has an unique additive gradient structure and uses forget, input and output gates to selectively forget or retain information from previous timesteps. In this architecture, the input gate takes in sequences of information or data and updates the cell. The output gate controls the flow of information leaving one LSTM cell to another. Lastly, we have the forget gate which controls the amount of information propagated to the next block. It uses the sigmoid value to decide this amount, a sigmoid value closer to one means that more historical information is stored, while a sigmoid value closer to zero means that less past information will be saved by the LSTM unit, thus "forgetting" or dropping parts of it. This is the main difference to a standard RNN, as the output gates of a LSTM network do not need to have all historical memory stored. An important takeaway here is that LSTM cells have in turn more parameters than RNNs, even when their architectural composition (number of layers/units) is the same. This number increases significantly with the dimension of input features and hidden units.

Now that we have a deeper understanding of the inner workings of AI concepts and various ML models, we can shift our attention to the actual use of AI/ML technology. In the next subsection we explain why ANNs are good candidates in handling data within several industrial use cases and also why their resource-targeted optimization is necessary in real-life contexts.

3.2 Intelligence at the Edge

First coined in 2011 (Kagermann et al., 2011), Industry 4.0 encapsulates the digitization and improvement of traditional industrial processes, as a consequence of recent advancements in technology. *Cloud computing*, which virtually offers unlimited computing power and storage, *mobile computing*, which facilitates wireless communication between devices, *Internet of Things* (IoT) characterized by the large array of cooperating objects such as general-purpose devices, and *Big Data* (BD), which represents large, growing volumes of structured or unstructured data collected from multiple sources, are considered to be key enablers of the fourth industrial revolution (Lee and Lim, 2021). These factors allow for continuous collection and analysis of physical phenomena via embedded and standalone sensors.

Some of the industries that have experienced a high impact due to these changes are manufacturing (Zheng et al., 2021, Rai et al., 2021), healthcare (Aceto et al., 2020) and logistics (C. S. Tang and Veelenturf, 2019). Under the IoT umbrella we find wearable IoT, defined by Hiremath et al., 2014 as a technological infrastructure that manages to interconnect wearable sensors with the goal of monitoring different human factors (health, wellness, behaviour) where the collected data can improve everyday quality of life. Wearable IoT devices represent only a small part of the available *sensing technology*, an overarching term for all devices that have the ability to acquire information by sensors. This type of technology supports the development of smart systems characterized by actionable intelligence, often registering the use of AI/ML technology. In fact, IEEE Innovation at Work predicts that AI and industrial IoT technologies will both be positioned in the top of trending topics of 2023 (IEEE, 2022). Wearable IoT and AI are highly prevalent topics in the health and manufacturing sector. A review by Mazzei and Ramjattan, 2022 highlights the primary topics and most common ML techniques pertaining to Industry 4.0 literature, indicating a trend towards neural network architectures used to improve production for industrial machines. A lot of sensor-collected data is used for inference

applications (Van Nguyen et al., 2021). Particularly in the health sector, we find many AI/ML-based applications created with the goal of performing otherwise human-dominated tasks, such as assisting clinician decision-making and treatment planning by predictive analytics (Siccoli et al., 2019), expertise assessments (Fard et al., 2018, Watson, 2014), drug development (Bannigan et al., 2021, Takagaki et al., 2010), detecting fatigue (Bai et al., 2020), stress (Garcia-Ceja et al., 2015, Plarre et al., 2011) and depression (Little et al., 2021), among others. There is however some skepticism towards interleaving AI capacities into an established system, on one side caused by the lack of a shared data infrastructure between organizations (Panch et al., 2019), and on another side due to different interests of the involved actors, such as privacy for users and explainability for practitioners (Shaw et al., 2019). Computational resources are also listed as a challenge, since health care organizations might not have the funds to secure resources or properly store and process data (Shaw et al., 2019).

In recent years, workplaces across industries make use of sensing technology paired with predictive technology in order to protect and promote health according to specific work group needs (Patel et al., 2022). Such an example is reducing the number of work-related accidents by various means of monitoring, tracking, and supporting workers (Svertoka et al., 2021). In a survey by the International Labour Organization (ILO), it is found that over 2.5 million people die from work-related accidents, and around 374 million suffer nonfatal injuries each year (International Labour Office, 2019). Only economic losses due to work-related accidents are estimated annually at around 4 percent of the global gross national product (Leppink, 2015). In order to avoid these repercussions at the individual and organizational level, it is important to discuss possible causes. In an infographic (Christ, 2016) we find that 60-80% of all work-related accidents are considered to be the result of different stress-induced manifestations having a negative impact on the workers' ability to safely perform duties. Health promotion and prevention programs are designed to boost productivity and reduce health-related costs and worker absenteeism (Hillier et al., 2005). But measures which

rely on self-reports are subjective, unreliable and may even be considered burdensome by those that perform them (Plarre et al., 2011). To overcome these challenges, latest advancements move towards a fusion between sensing and predictive technology, making health monitoring in workers more accessible and more objective (Bangaru et al., 2022, Escobar-Linero et al., 2022). Although we have centered this discussion around health monitoring applications it should be noted that it is just one of many perspectives from which we could have started this conversation. Time-series is a data type resulting from sensor readings under many other scenarios. Various time-series-based applications found across industries like finances (Ghasemzadeha et al., 2020), agriculture (Bína et al., 2022, G. Liu et al., 2022) and arts (Huang and Wu, 2016) show how deep learning technology can assist, take over otherwise cumbersome tasks, or even generate new artifacts.

Contrasted to human capabilities, computers are able to store, access, find hidden patterns and correlations in large amounts of data at a faster pace. On the other hard, deep learning models are typically computationally-intensive, have high memory, power and energy requirements, which not all types of computers can meet. Many startups and larger companies adopt cloud computing (H. Liu, 2013), that is, processing data on a remote server, as it allows them to harvest benefits such as *"elastic"* capacity which gives the illusion of indefinite resources (Buyya et al., 2010). But not all tasks can be efficiently performed this way. Limited network bandwidth can limit or restrict the transmission of essential training sensor data to the cloud. These bandwidth-induced delays can negatively impact interactions between people and applications, reducing usability (Satyanarayanan et al., 2009). Sending data to the cloud also introduces privacy and security risks such as exposure to Denial of Service (DoS) attacks, injection of cloud malware and communication interception, which is highly concerning for applications that handle sensitive data (Parikh et al., 2019). Parikh et al., 2019 contrasts traditional cloud computing to fog and edge computing and motivates its replacement. In recent years, developers rely on various optimizations to allow deep model training and inference place-

ment on or near edge devices (Chen and Ran, 2019). Computing on edge devices allows continuous operation, reducing latency as network availability no longer becomes a hinder, and allowing sensitive data can be stored and processed locally, thus shielding users from unfortunate or malicious events. The intersection between AI and edge computing has been coined *edge intelligence* (D. Liu et al., 2022). D. Liu et al., 2022 also speaks of the computational gap separating complex AI models from less-capable edge devices. Dhar et al., 2021 enumerates four critical resource constraints that challenge edge learning: processing speed, memory, power and energy consumption. Accommodating the resource requirements of AI models on low-power devices is currently a high-interest area of research (Chen and Ran, 2019), and also a main consideration in our thesis.

Time-series data is outputted by many ubiquitous devices, from wearables to industrial sensors. This highlights the need for further research in optimizing time-series processing technology.

3.3 Sensors and Time-series Data

The IoT paradigm together with BD, are closely related to AI since they represent ways of obtaining heterogeneous input data necessary for training various ML models (Kaplan and Haenlein, 2019). With an increase in sensor technology popularity, data often takes the form of a sequence of quantitative observations which have to be taken at consecutive points in time for one or several entities/processes (Pal and Prakash, 2017). This data is either irregularly or regularly observed, so the space between the point recordings can either be equal or not. In Pal and Prakash, 2017 we find a distinction between three categories: *cross-sectional data*, *time-series data* and *panel data*. Cross-sectional data are observations from several individuals which are taken at the same point in time, while time-series data consists of quantitative observations on one or more measurable characteristics of one individual entity/process taken at multiple points in time. Panel data is obtained by observing multiple entities over multiple points in time.

Pal and Prakash, 2017 describes characteristics of time-series data and shows how any data point can be represented by the presence or absence of general trend, seasonality, cyclical movements, and irregularity. Cipra, 2020 describes *trend* as long-term increase or decrease in values of a time-series, *seasonality* as periodic changes in time series that repeat at given intervals, *cyclical* movements as fluctuations around the trend where increasing phases alternate with decrease phases, and *irregular* (or *residual*) variations as formed by random fluctuations of time series, often coined white noise. Another important practice is checking for outliers. Real monitoring applications where data is acquired through sensing technology is prone to sensor faults. Some of the most commonly seen faults are outliers or spikes, missing data, noise, "stuck-at" or constant values (Teh et al., 2020). Low battery, environmental or malicious interferences, uncalibrated sensors and unstable wireless/network connection are some of the factors that can cause such faults (K. Ni et al., 2009, Y. Li and Parker, 2014). If the collected data shows signs of missing or abnormal values, it is common to use anomaly detection methods in order to remove or repair inconsistencies. Further preprocessing of time-series data involves other steps, such as employing a windowing approach, shifting across a number of values.

All in all, we can say that AI/ML optimization is an area of research which deserves more attention as its utilization has extended across many industries. We underline that the deployment of AI within the edge computing paradigm with high real-time performance expectations and high accuracy requirements is a pressing Industry 4.0 matter, and finding an effective solution to this has many practical implications. To connect all dots, we remind our readers that the final goal is finding a generally-applicable set of optimizations in order to improve AI/ML-augmented systems whom employ the use of sensing technology, which often translates to time-series analysis. These three background sections contribute to the motivation behind our thesis. Now that we have a wider picture of the research field, the technology and the type of data of interest we can continue the discussion on resource-efficient techniques by exploring prior contributions to this research area.

4 State of the Art

We want to capture the essence of current solutions, and find those that are commonly used across considered architectures, which will serve our intuition while designing the final framework. Exploring current literature on the topic of resource-focused optimization is also necessary to identify neglected areas of research, and so our take on research gaps and thesis contributions can be found at the very end of this section. Inspired by Kofod-Petersen, 2012, we embark onto a three-stage review process, first by planning, then performing and lastly reporting the literature review. The overall process flow is pictured in figure 2, starting from the leftmost column and finalizing the review by following all the steps leading to the rightmost column. In short, we develop a literature review protocol which facilitates a systematic approach for relevant literature collection. From this collection of papers we will extract core knowledge, necessary in answering posed research questions and identifying potential research gaps.

4.1 Planning and Performing the Literature Review

Conducting a well-planned research literature review increases reproducibility. The goal of the planning phase is developing a literature review protocol, which provides step-by-step instructions. We begin by setting some objectives. The end goal with our literature review is grasping current trends, tools and existing technologies that aim to improve time-series data-processing ANNs, and as a result aid deployment and inference on resource-constricted devices. In order to obtain more knowledge about the above-mentioned, a practical idea is to split this exercise into smaller and more manageable parts, around which we formulate research questions. In short, what we are interested in is finding which deep learning models, devices, performance metrics and techniques have been given attention in previous works. We can thus reformulate our initial review goal by considering all of its parts. By doing this we end up with four research questions, as follows:

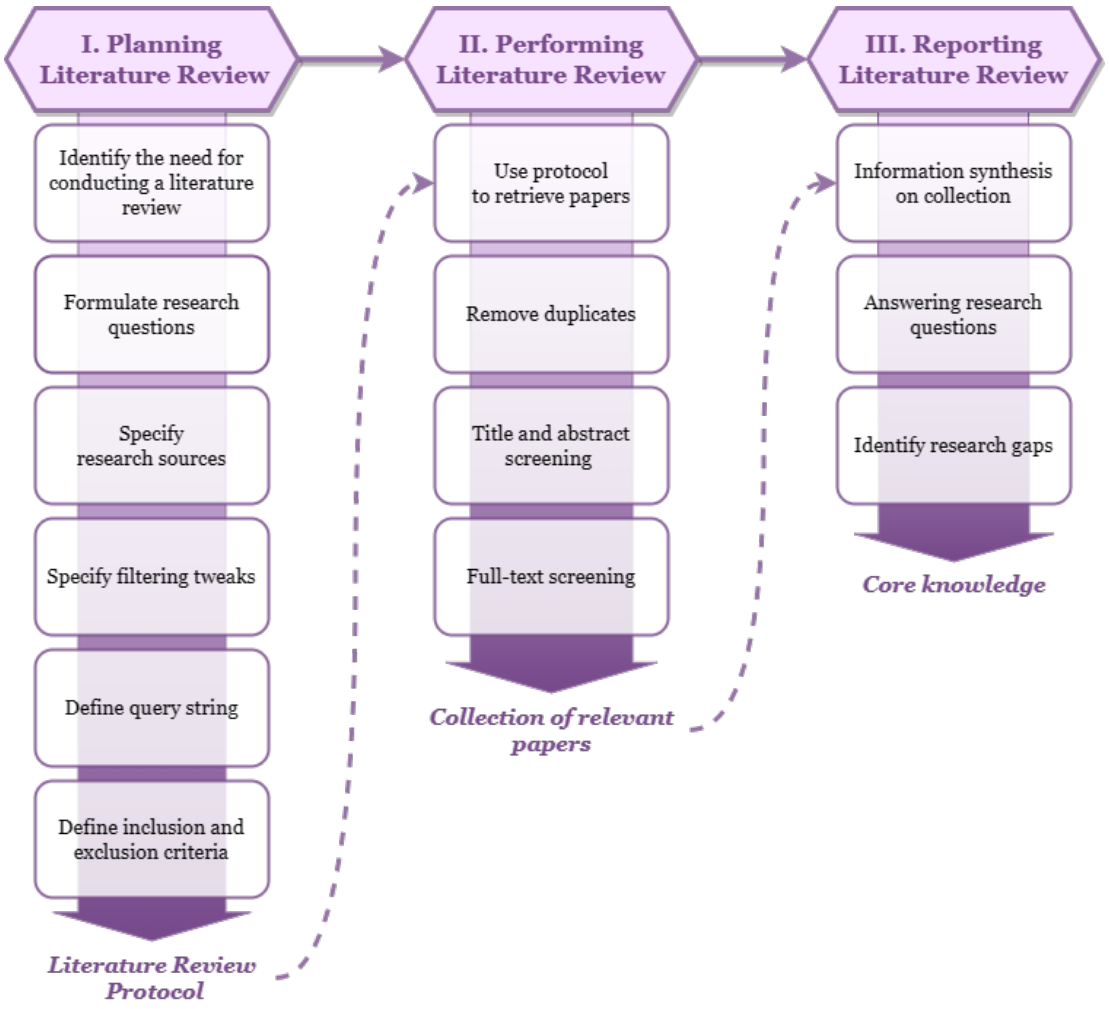


Figure 2: Systematic Literature Review Process.

***Research Question 1* (RQ1)**

Which **ANN architectures** typically process sensor data?

***Research Question 2* (RQ2)**

Which **devices** are typically targeted for deployment and inference?

***Research Question 3* (RQ3)**

Which device **resources** are typically impacted by inference and how are they measured?

***Research Question 4* (RQ4)**

Which **optimization** techniques are typically employed to realize deployment and inference?

Answers to the above-mentioned questions provide insights in the general statistics around the collected articles, helping us get a picture of the overall trend. But more importantly, these answers allow us to find common optimizations that can be applied across architectures, or discover papers that attempt doing so. Thus, apart from understanding status quo, the answers to all of these questions are meant to support finding a general way of optimizing neural networks which can be applied across architectures and complexities. From a different perspective, we see that these answers also help uncover whether this area of research has been sufficiently explored.

The next step is specifying literature sources. We choose to sample publications from three different sources: the ACM Digital Library[1], IEEE Xplore[2] and lastly Oria[3], which is a cluster of scientific databases. We will narrow our search by allowing only publications from January 2018 up to January 2023, thus keeping the pool of research papers closer to current date ($\approx$ 5 years). Right away, we filter out papers not written in English, and those that are not accessible. To define our query string, we group topic-related keywords together by synonymous relations. In

[1] **https://dl.acm.org**.
[2] **https://ieeexplore.ieee.org**.
[3] **https://oria.no**.

this query string the $\vee$ symbol stands for the logical *OR* operator and the $\wedge$ symbol for the logical *AND* operator, and we assume that the role of wildcards are known. The $\neg$ symbol stands for logical *NOT* operator, which is used to filter out papers which contain the terms in this category. These terms have been chosen by refining the query string multiple times in order to reflect our thesis goal. Keep in mind that the query string format may vary as libraries use different search conventions, however the same logic applies.

Query String	(inferenc* $\vee$ "post training" $\vee$ "post-training" $\vee$ deploy*) $\wedge$ ("neural network" $\vee$ "deep learning") $\wedge$ ("light-weight" $\vee$ "lightweight" $\vee$ resource* $\vee$ "optimi*") $\wedge$ (edge $\vee$ device*) $\wedge$ ("time-series" $\vee$ "time series" $\vee$ sensor $\vee$ wearable) $\wedge$ ($\neg$ federat*) $\wedge$ ($\neg$ distribut*) $\wedge$ ($\neg$ accelerator)

The query string searches through document abstracts only, except for the $\neg$ terms, which target only document titles since we do not want to filter away papers that employ a combination of optimizations. Note that federated learning, computational distribution and the use and improvement of accelerators will not be the focus of our thesis, which is the reason we filter away papers where these are main contributions. We will however keep a short mention of these in section 4.2, as they are popular in minimizing resource utilization on edge devices. To complete the literature review protocol, we must incorporate inclusion (IC) and exclusion criteria (EC), which help us filter out low-relevance papers in the next stage of the process. We are suppressing all inclusion criteria into one criteria (IC). Additional filtering is facilitated through exclusion criteria, where papers that fulfill at least one exclusion criteria are removed from the pool. Inclusion and exclusion criteria applied in this review can be found in table 1.

Performing the review begins with the initial search for relevant papers using the query string together with the filtering tweaks specified in the protocol. This returns a set of 299 published papers. Out of the total, 21 stem from Oria, 210 from IEEE Xplore and 68 papers from

Inclusion and Exclusion Criteria	
Type	Criteria description
IC1- IC4	The paper concerns itself with or helps answering any of the research questions from above (RQ1 - RQ4).
EC 1	Papers focused solely on distribution, decentralization, scheduling, offloading of computations across devices, privacy concerns, fog or federated learning.
EC 2	Papers focused solely on hardware-related optimizations.
EC 3	Papers where no resource-efficient optimizations have been presented or used.

Table 1: Inclusion (IC) and exclusion (EC) criteria.

ACM Digital Library. After removing duplicates, we perform the first screening which will reduce this number substantially, using the inclusion and exclusion from Table 1. For the first screening, only titles and abstracts are considered. After the screening, a total of 215 papers are eliminated due to their low level of relevancy (or if any exclusion criteria has been fulfilled), and 84 remain in this intermediary set of papers. We then perform the full-text literature screening of the primary studies, extracting core information necessary for answering the research questions proposed earlier.

Extracting core knowledge is facilitated through tagging each paper with the type of model architecture, the training and deployment environment, the set of performance and evaluation metrics used, the applied optimizations, and the scenario or use case in which it is used. There are some exceptions where not all variables are present, such as in the case of surveys where device specifications and metrics are not explicitly specified. After the full-text screening, we end up with a final collection of 66 papers with high relevancy. We make use of this information to answer the previously posed research questions in the next subsection.

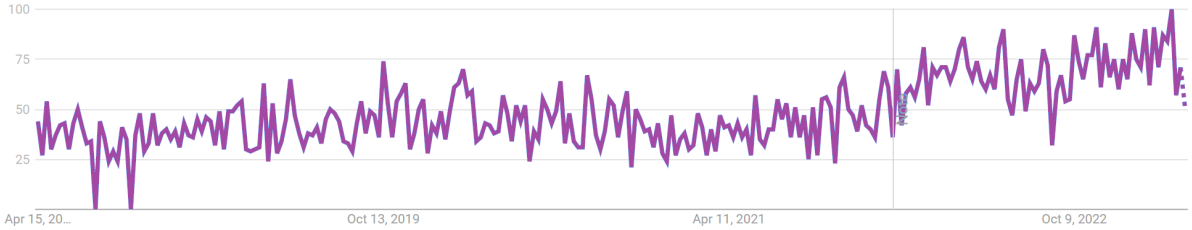


Figure 4: Google Trends search popularity plot for "machine learning optimization" query (2018-2023).

4.2 Main Findings of the Literature Review

A first step into reporting findings from the collected literature is capturing the overall trend for the research area. This is better observed by plotting the number of papers against the publishing year, as shown in figure 3. For our small sample of relevant papers, we register an almost consistent increase in interest for resource-efficient ML research during the last 5 years, with higher occurrence as one gets closer to current date (note that the 2023 bar only considers the first month of the year).

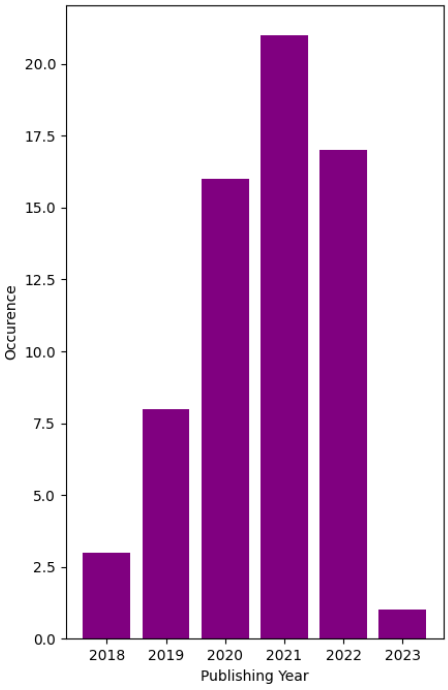


Figure 3: Number of published papers per year.

A possible explanation for this is that while deep learning technology applications have gained more and more popularity, the issue of ML resource optimization is just currently gaining attention in the research space. We observe that "machine learning optimization" as a search query also has a similar distribution when plotted by Google Trends (see figure 4), reinforcing our previous statement. Based on

literature review findings, multiple technological advancements within various industries are represented by real-life applications where sensing technology is coupled with the predictive ability. Some of the most prevalent use cases are found within the health sector, it examples such as identifying various heart anomalies (Burrello et al., 2022, Sakib et al., 2020, Tesfai et al., 2022, Ukil et al., 2021, Sun et al., 2021, Faraone and Delgado-Gonzalo, 2020, Faraone et al., 2021, J. Wu et al., 2019, Wong et al., 2022a, Burrello et al., 2021, Mirsalari et al., 2021), blood glucose prediction (Zhu et al., 2021), detecting stress (Ragav et al., 2019) and other mental states (Dey and Roy, 2020), epileptic seizures (S. Zhao et al., 2021), fall detection (S. Li et al., 2022, Putri et al., 2021), and gait disorders in elderly (S. Y. Tang et al., 2019), among others. We also find a large series of lightweight applications related to human activity recognition (Bian et al., 2022, Daghero et al., 2022, Coelho et al., 2021, X. Cheng et al., 2022, Bohra et al., 2021). A smaller set of applications are found within the transportation (Mishra et al., 2020, Sajjad et al., 2020, Hussain et al., 2021) and security (Zouridakis and Dinakarrao, 2022) domains. Some interesting but very domain-specific use cases are found for marine mammal recognition systems (Y. Zhao et al., 2022), recognizing ocean wave patterns (H. Wu et al., 2020), coffee plant disease detection (De Vita et al., 2020), natural hazard monitoring (Meyer et al., 2019) and fire detection (Thomson et al., 2020).

Our efforts now will be directed towards answering the research questions from section 4.1 based on extracted core knowledge.

4.2.1 Common neural network architectures (RQ1)

By tagging papers with the type of model architecture during the full-screening, we are able to determine which model architectures attract most interest. Note that some papers can have several tags, if more than one architecture has been considered. There are also models created by fusing together two different architectures (here, we will use the + symbol to represent fusion). As shown in figure 5, CNNs are by far one of the most popular type of model, followed after by the LSTM. We

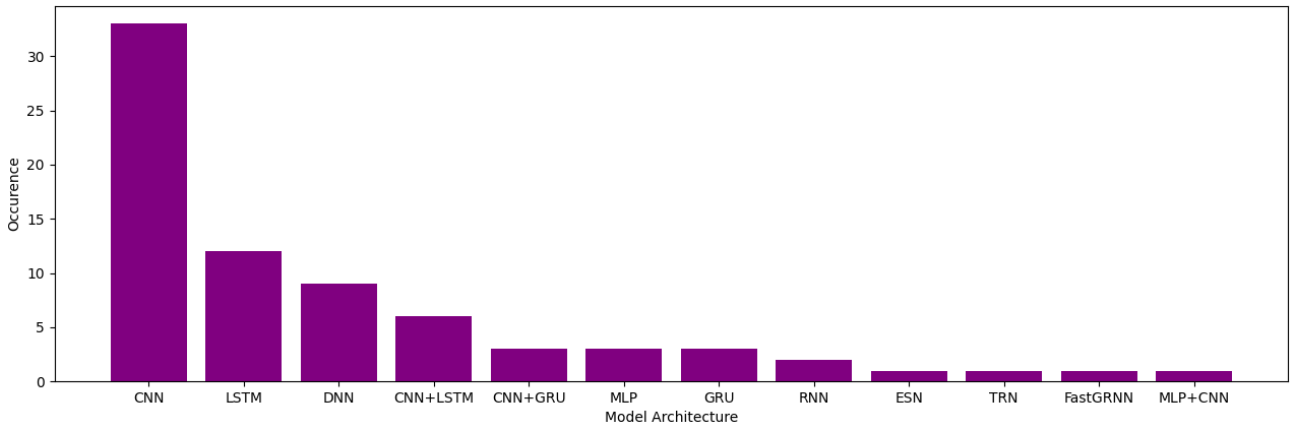


Figure 5: Popularity of model architectures.

note that fusion architectures which contain a CNN are also quite popular, especially is combination with LSTMs. Temporal Convolutional Networks (TCNs) and other variants of the CNN have been tagged as such. In this plot, DNN refers to papers which include optimizations that can be used across several architectures, or where authors only refer to models as *deep neural networks*. Although the TRN (Temporal Relation Network) contains some convolutional layers, we will process it individually, since it is a complex architecture that also has a temporal module with a self-attention mechanism. FastGRNN is a gated recurrent network architecture, very similar to the GRU architecture. According to our findings, the most popular model architectures used for processing time-series or sensor data are CNNs and LSTMs. It should be noted that some of the MLP papers were removed in the filtering process, not because of their lack of optimizations for resources, but due to the fact that their optimizations involved topics from the exclusion criteria.

4.2.2 Common deployment targets (RQ2)

A common theme in our literature is deployment onto low-power or even battery-free edge environments. Thanks to deep learning optimizations, off-the-shelf, small-footprint devices can be turned into deployment and processing environments for various inference tasks. By tagging our

papers with the type of device and the technical specifications, we observe that IoT end-nodes are typically based on general purpose central processing units (CPUs) due to their high programmability, low power consumption and low cost. Edge devices that occur in more than one paper are the Raspberry Pi 3B/3B+/4B, NVIDIA Jetson TX2, Nexus 5, and other smaller hardware pieces that include processing units from either the ARM-Cortex M or the ARM-Cortex A family of microcontrollers and respective processors. In the ARM-Cortex M microcontroller family we find variants such as M0+/M3/M4F/M4/M7 while in the ARM-Cortex A processor family we find variants with A53/A57/A73 cores. We note that most of these are used in low-cost, portable systems. For example ARM-Cortex M family devices have a flash memory that is limited to few MBs, and a SRAM size that ranges from tens to hundreds of KBs only. Some of the devices we have named previously belong to these families (e.g. Raspberry Pi 3B+ has a quad-core ARM-Cortex A53 with an operating frequency of 1.4GHz), however the actual device specifications are not always specified in the papers. One important takeaway here is that many experiments run on different versions of Raspberry Pi (e.g. S. Liu et al., 2020, S. Liu et al., 2021, Chauhan et al., 2018, Sakib et al., 2020, Zouridakis and Dinakarrao, 2022, Ragav et al., 2019, Sajjad et al., 2020).

4.2.3 Common resource constraints (RQ3)

By answering RQ3, we should get a picture of what type of resources we want to preserve once our models are deployed, but also understand how these and prediction performance are measured under various use cases. Another goal here is understanding how the two categories are contrasted against each other. This is critical when setting up an analysis with respect to accuracy and resource utilization. We aid this process by tagging the core literature with the most common evaluation metrics employed in scenarios where resource utilization and model accuracy have been measured.

Resource constraints given by hardware architecture, such as processing speed, memory constraints, energy and power consumption should be main considerations when designing an AI/ML model for on-device inference (Dhar et al., 2021). ML development that considers reliability, security, and other social aspects has also attracted attention (Shafique et al., 2018), however these characteristics are beyond the scope of our thesis. Apart from model performance in terms of number of good predictions (i.e accuracy), we are interested in how we can quantify resource performance at inference for inference time, memory utilization, energy and power consumption. We have found six categories of metrics relevant to our thesis goal. While we have attempted to match the metrics found in literature to one of these six categories, there might be some overlaps or ambiguities. Some of this stems from the fact that some papers do not make a clear separation between resource performance and accuracy performance, and may use a combination of these to compute overall performance. There will be thus some assumptions on our part as to where they can be inserted in our taxonomy:

- **Prediction performance** Depending on the task type, there are various ways of quantifying predictive performance. Most papers that we have considered in this review use accuracy as a prediction performance metric. Accuracy calculates the fraction of the total number of correct predictions divided by the total number of model predictions. We note that the term *accuracy* is also used in some papers (as in this paper) to refer to overall predictive performance. However, given an imbalanced data set or a situation where getting false predictions is very costly, this metric may not be a good choice as it can over-represent the majority class. In such cases we see the use of balanced accuracy (Daghero et al., 2022) and/or other metrics such as precision, recall and the harmonic precision-recall mean (Tesfai et al., 2022). In quantifying model performance, another frequent occurrence is a combination between accuracy, precision, recall and F-score. For example, Thomson et al., 2020 uses F-score, precision, accuracy together

with true and false positive rate to compare performance of various architectures. Under the context of fall detection, Putri et al., 2021 uses accuracy, recall, precision, F-score and area under the curve (AUC) in comparing performance between reccurrent architectures. De Vita et al., 2020 uses accuracy, precision and recall to compare a full-sized model to four optimized versions. In Zebin et al., 2019 F-score is found to be a better indicator for cases where recall and precision are uneven. Another way of quantifying inference error is calculating the root mean square error (RMSE) and mean absolute error (MAE), where smaller values are desired (Zhu et al., 2021, Nizam et al., 2022). A wide variety of error measures are also used in Peluso et al., 2021 as evaluation metrics.

- **Inference latency** Shuvo et al., 2022 defines latency as the speed by which an inference engine can accomplish a complete inference. Latency is argued to be dependant upon the number of interdependent layers, number of neurons/computations per layer, latency of memory accesses, number of memory accesses and latency of the computing modules (Shafique et al., 2018). Inference time can be measured by simply timing the process from the inference request to the point where the prediction has been made (e.g. by subtracting timestamps), and the measurement unit is typically seconds or milliseconds. However, there seem to be cases where measuring inference time can take a different approach, such as by summing up the processing time of each layer in order to create an latency estimate (Yang et al., 2021), or using frames-per-second for video comprehension frameworks (Y. Cheng, Li, et al., 2020, Shafique et al., 2018, Y. Cheng, Huang, et al., 2020, Thomson et al., 2020). Others choose to also keep track the number of CPU cycles per inference, and use the ratio between cycles and MACs to quantify implementation efficiency (De Vita et al., 2020). Execution time can also be estimated from FLOPs (Mishra et al., 2020).

- **Memory utilization** The devices presented in subsection 4.2.2 are generally equipped with two types of memories, a volatile static

or dynamic random-access memory (SRAM or DRAM) and a non-volatile flash memory. In regards to the constraints posed by memory size, one should also keep in mind that the AI model does not only have to fit on the read-only memory (e.g. flash), but also on the read/write memory which contains mutable states during execution (Kumar et al., 2020). The non-volatile memory stores the actual code and static data such as learned parameters of the neural network. The evaluation benchmark in Profentzas et al., 2021 uses memory allocation for RAM and flash for quantifying model memory footprint. Flash usage is also used in Kumar et al., 2020 to check if compression allows models to fit onto devices, while RAM usage is estimated from the sizes of temporary variables.

The work by Wang et al., 2020 uses an estimation of the required memory using the number of neurons, layers, weights and size of data buffer to evaluate network size for further mapping onto the cache closest to the processing unit. To check memory consumption Y. Cheng, Li, et al., 2020 makes use of the htop tool in Ubuntu. In a survey by Lalapura et al., 2021 memory savings are calculated by using the number of parameters, but also by the actual size of the model (in kilobytes). Memory is listed as a quality metric for edge inference performance in Shuvo et al., 2022, who states that memory requirements are contingent upon the model size, the number of memory access, and memory types. In Risso et al., 2022 the actual model size (in kilobytes) is used to capture the memory use. The memory allocation analysis in Zebin et al., 2019 lists the number of nodes, their allocated memory, and their average execution time.

Memory consumption can also be estimated from FLOPs (Mishra et al., 2020). Memory footprint in Faraone and Delgado-Gonzalo, 2020 is computed by looking at the memory allocated for static data and RAM-allocated data. De Vita et al., 2020 uses memory allocation on flash and RAM as a metric for performance as well. After quantization, Meyer et al., 2019 uses the number of param-

eters to determine whether or not the model fits in SRAM, fulfilling the memory requirements. Peluso et al., 2021 uses RAM allocation as a memory footprint metric, categorizing it as an extra-functional metric.

- **Power consumption** Power consumption reflects the total power required to execute the inference (Shuvo et al., 2022). Power consumption is usually expressed in watts, such as milliwatt (mW) or microwatt (μW) (as seen in Wang et al., 2020, Y. Cheng, Li, et al., 2020, among others). To analyze the power consumption Y. Cheng, Li, et al., 2020 makes use of a DC-regulated power supply. In Bian et al., 2022 power consumption is measured by a power analyzer. As exemplified in Y. Zhao et al., 2022, the most common way of computing power consumption is by multiplying the current and the voltage on the running device, as it is connected to a power supply. According to Shuvo et al., 2022 power consumption (and implicitly energy) is highly impacted by the number of computations and off-chip memory accesses. In Faraone and Delgado-Gonzalo, 2020 the power efficiency is equal to the ratio between throughput and power, where throughput is equal to the number of operations divided by execution time.

- **Energy consumption** Energy consumption equals the rate of operations that an edge node can process per watt of power consumption (Shuvo et al., 2022). We observe that energy and power consumption are related, although not proportional. Energy consumption is usually measured in joules (as seen in Wang et al., 2020, Burrello et al., 2022, Daghero et al., 2022, Y. Zhao et al., 2022, Risso et al., 2022, among others). In Profentzas et al., 2021 energy consumption is measured based on the average electric current, the inference time and the operating voltage. The same formula is given by De Vita et al., 2020, where average energy consumption for each inference is equal to the product between the supply voltage, the average inference time and average absorbed current per millisecond. S. Liu et al., 2020 validated their frame-

work efficiency by measuring energy consumption by means of using an external power monitor. The energy efficiency estimation in S. Liu et al., 2021 considers also parameter and activation arithmetic intensity.

Apart from the limited storage issue seen above, the number of read accesses to these types of memory should also be kept minimal, as it consumes energy (Meyer et al., 2019). In an overview by Shafique et al., 2018, we find that one challenging aspect to the power and energy efficiency of a constricted device is that each multiply and accumulate operation (MAC) is normally coupled with three memory reads and one memory write, which even in the case of simpler models would result in millions of memory read and write operations. According to Shuvo et al., 2022, energy efficiency is impacted mostly by the number of memory accesses, model size and other computation needs. The study by Lu et al., 2019 shows that energy consumption is expected to affect a user's quality of experience, and so they design an energy function that takes in parameters such as number of nodes, number of layers, the CPU and RAM capacity. While evaluating energy consumption on microcontrollers, Coelho et al., 2021 lists the current value and mean current value when generating one classification per second as metrics. Energy consumption can be estimated from FLOPs (Mishra et al., 2020). In their experiment Sajjad et al., 2020 makes use of an USB detector to calculate various energy-related parameters, such as current, voltage, power and energy.

- **Model complexity** Shuvo et al., 2022 considers the size of the model as an important metric in edge inference evalution, since low-complexity, compact network architectures are better suited for edge deployment. They state that aside from actual hyperparameters (number of layers, neurons, activations, losses and so on), one can also use the number of learnable parameters since it reflects memory requirements. To compare models with similar complexities, Cerina et al., 2020 adjusts the number of active

neural units by making sure that the number of parameters is approximately the same for all considered architectures. The paper by Burrello et al., 2022 shows that model complexity can also be expressed by the number of parameters and MAC operations. The number of parameters is also used as a complexity proxy in comparing baseline models to their compressed versions (Xiong et al., 2020). We observe that the number of parameters, together with the average bitwidth and actual model size (in bytes) are also used for getting a better understanding of the compression effect on model layers (Sun et al., 2021). Coelho et al., 2021 argues that MACs are a good metric for understanding CNN complexity, since they account for a large part of the operations, and uses these together with the model size (in kB) and the number of parameters to map models of various complexities. For CNNs, Wong et al., 2022a uses MACs for quantifying algorithm complexity.

We can clearly differentiate between accuracy-related metrics and resource-related metrics. Members belonging to either of these two categories are also often compared against each other, in order to capture the trade-off between accuracy and resource utilization. Our analysis between the full-sized and optimized model should also aim to capture these relations. In contrast to using a combination of metrics for one specific goal, some papers evaluate performance in terms of trade-offs between metrics, typically mapping accuracy against a resource performance metric. One such example is found in Daghero et al., 2022, a study which identifies two sets of Pareto-optimal architectures considering accuracy versus memory occupation and accuracy versus number of cycles per inference. Pareto fronts considering accuracy versus number of parameters are explored in Risso et al., 2022 for TCN models. Chandna et al., 2023 considers the trade-off between accuracy and model size for binarized models. A similar approach where comparisons between models resulting from several levels of applied optimizations using accuracy versus model size is found in Yu et al., 2022. We conclude that the presence of a single metric might not always be sufficient

Key Performance Indicators

ID	Name	Description	Binary formula	Multi-class formula (C_1, C_2, C_3)
KPI_1	Precision / Macro average precision	Correctness of the proportion of identifications in a model	$\frac{TP}{TP + FP}$	$\frac{KPI_{1,C1} + KPI_{1,C2} + KPI_{1,C3}}{3}$
KPI_2	Recall / Macro average recall	Correctness of the proportion of actual positives being correctly identified by the model	$\frac{TP}{TP + FN}$	$\frac{KPI_{2,C1} + KPI_{2,C2} + KPI_{2,C3}}{3}$
KPI_3	Harmonic Precision-Recall Mean (F1) / Macro average F1	Combines precision and recall	$\frac{2 \times KPI_1 \times KPI_2}{KPI_1 + KPI_2}$	$\frac{KPI_{3,C1} + KPI_{3,C2} + KPI_{3,C3}}{3}$

Figure 6: Model prediction performance indicators.

to effectively measure performance in terms of either accuracy or resource use, but their combination might spark new insights about the overall performance, and help us understand their relationship better.

Based on everything we have seen in this subsection, it is possible to distinguish some common patterns in the way that researchers choose to evaluate performance for either accuracy or resource use. We can use this to motivate our own performance indicator choices. These are called *key performance indicators* (KPI). These are critical components to our envisioned PoC because they are used to evaluate both the full-sized models and their lightweight counterparts. First we will construct a list of performance indicators that reflect efficiency in terms of successful predictions for classification tasks (KPI_1 - KPI_3, figure 6). The choice is justified by information extracted from the core knowledge here and other readings where classification tasks are tackled (Ghosh et al., 2022, Uddin and Canavan, 2019). Note that for the multi-class classification task we adapt our formulas so that predictions from several classes are considered into one metric. That is, we will average the previously mentioned metrics for all classes, treating each class equally. This is because in the evaluation phase we will choose an evaluation inference set where each label appears equally, but also because all classes are equally important to detect. When calculating macro-precision and macro-recall in the multi-class case, we substitute the TP, FP and FN values as explained in section 3.1.

Key Performance Indicators

ID	Name	Description	Formula / Implementation
KPI_4	Inference Latency	Measure speed during one complete inference request	Inference time in milliseconds (ms)
KPI_5	Parameters	Estimates memory requirements and model complexity	Number of learnable parameters (fullsize)/ Estimated from tensor shape products (optimized).
KPI_6	Compressed file size	Estimates necessary memory requirements for a compressed model.	Compressed model size in kilobytes (kB)
KPI_7	Non-volatile memory	Portion of space allocated for non-volatile storage of model	Actual model size in kilobytes (kB)
KPI_8	CPU usage	Estimates system-wide CPU utilization as a percentage	Uses cpu_percent from psutil package
KPI_9	Power consumption	Quantifies the power needed to perform a complete inference request	$KPI_9 = V * I$ (in W) (I = current in A, V= voltage in V)
KPI_{10}	Energy consumption	Quantifies the energy needed to perform a complete inference request	$KPI_{10} = KPI_9 * KPI_4$

Figure 7: Model resource efficiency indicators.

With a better understanding of common resource-related metrics used across current literature, we will also update our KPI list with resource utilization indicators based on the findings from this section. KPI_4 - KPI_{10} assess resource utilization efficiency in the deployment environment in terms of inference latency, memory utilization, power consumption, energy consumption and model complexity. Their identification number, metric description and the formulas needed for calculations can be found in figure 7. Note the addition of KPI_6, which has been recommended when comparing full-sized and optimized models in Tensorflow (Tensorflow, 2022).

4.2.4 Common optimizations and techniques (RQ4)

This subsection introduces concepts linked to resource-focused optimization through the lenses of current research. Our goal with this subsection is to provide a high-level taxonomy of the most common optimization techniques. This process is facilitated by tagging core literature with found optimizations.

While training a deep model may require millions of parameters to be refined over and over again, deployment and inference are also expensive with regards to the previously-listed performance metrics necessary to complete a forward pass (i.e inference). Shuvo et al., 2022 enumerates three popular use cases in deep learning: 1) training and inference on the cloud, 2) training and inference on the edge and 3) training on the cloud and inference on edge. In this subsection we will consider optimizations that can be applied to models that fall within the third category, with a focus on edge inference.

We find two research directions: one towards more efficient hardware platforms that are able to efficiently store and run complex ML algorithms and another towards optimizing ML models so that we reduce the resource requirements of the target device. This review sets a bigger focus on the latter. One reason for this is that customizing hardware accelerators has high manufacturing costs and limited generalization, while the software itself is more flexible to optimize and less expensive (D. Liu et al., 2022, Thompson et al., 2020). The study by Thompson et al., 2020 shows a clear sustainability issue with scaling deep learning computation by scaling up hardware, which leads to environmental and monetary constraints. As deep learning computational constraints are getting more and more difficult to fulfill by current technology, hardware specialization becomes less popular (Thompson et al., 2020). D. Liu et al., 2022 states that innovative deep learning techniques will remain "key ingredients" in supporting edge intelligence, when contrasted to hardware optimizations.

We find a real need for ML models to be optimized in order to tackle the memory and computational constraints of the targeted devices. These

optimization techniques can be applied at different stages in the model development process. Note however that some optimizations can cross these temporal categories, but also that general practice involves using a set of techniques falling within either of the aforementioned categories. We present the following three-folded taxonomy over optimizations:

4.2.4.1 Hardware Optimizations, Distributed Computing

When it comes to hardware optimizations, current efforts explore paths towards more energy-efficient computing architectures (Van Nguyen et al., 2021). According to Cerina et al., 2020, the most promising approaches to improve the performance of ultralow-power processors are parallelism, low-power fixed-function hardware accelerators, memory access optimization, and near-threshold technology. In a overview of current optimization trends by Shafique et al., 2018 we also find listed designing specialized hardware accelerators, using hardware capable of AI acceleration and near threshold computing. Thus, hardware can be improved by allowing parallel implementation of multiple cores, as seen in the case of microcontrollers (Wang et al., 2020). Gupta et al., 2020 states that process-in-memory architectures provide significant parallelism while reducing data movement between the memory and processing cores. Their study implements row-parallel operations internally in a crossbar memory. In Van Nguyen et al., 2021 we see a DL implementation over memristor crossbars instead of the typical Von Neumann architecture. The way that memory is allocated to different processes on hardware is also an important consideration in Kumar et al., 2020. Enabling single instruction/multiple data (SIMD) operations are optimizations which too enable concurrent core execution (Sajjad et al., 2020, Schneider et al., 2020). Constricted IoT systems are sometimes augmented with Field-Programmable Gate Arrays (FPGAs). Shuvo et al., 2022 defines FPGA as a fine-grained reconfigurable architecture with programmable logic blocks and configurable interconnections, that can be programmed through hardware description languages. These systems thus benefit from more computational power, some studies focus-

ing on generating specialized accelerators that optimize their performance (Qian et al., 2022, Bahrebar et al., 2021). Wong et al., 2022b also uses FPGA for inference by partitioning allocated memory. Parallelism aided by FPGA hardware accelerator is also employed in S. Y. Tang et al., 2019. Neural network processing units (NPU) can also be present in hardware, aiding in processing computational loads (Y. Cheng, Li, et al., 2020). AI cores (S. Li et al., 2022), or specialized embedded AI devices (Putri et al., 2021) are also present in other studies, accelerating computations. The work in Yu et al., 2022 is aided by GPU acceleration.

Distributed computing is one way of reducing the high processing costs is enabling fine-grained latency/energy-aware distribution of computations across the system edge-to-cloud stack (Shafique et al., 2018). In Shuvo et al., 2022, edge offloading is considered one major edge AI component that boosts edge inference by offloading the computation-intensive task to the cloud or partitioning it. For example Nizam et al., 2022 improves the model over time by offloading to cloud. A disadvantage is that this process can have a data transmission bottleneck. In Yao et al., 2019 this process is improved by compressing and then reconstructing transmitted data at the edge server by the use of a decoder.

4.2.4.2 Training-time Optimizations

Probably the most obvious optimization for resource utilization at training-time is the use of smaller models, or improved architectures. One way of keeping models small is input representation. Lower sampling rates and reducing the number of variables helps for example in reducing filter size for CNNs (Coelho et al., 2021). Other techniques include using principal component analysis for input reduction (Wei and Radu, 2019) and converting to other data types, for example by converting video to time-series (Y. Cheng, Li, et al., 2020, S. Li et al., 2022), or converting time-series to two-dimensional images (J. Ni et al., 2022) thus leading to simpler and more energy-efficient architectures.

One dominant strategy to reducing model complexity and controlling memory footprint is to utilize network architecture search (NAS) mechanisms. NAS explore a large design space comprised of combina-

tions of layers and/or hyper-parameter values, choosing the best solutions for optimizing a specific loss/cost metric. The work by Wong et al., 2022a shows how best-performance and low-power model designs can be found through Monte Carlo simulations. Other implementations make use of multi-objective optimization methods that consider both accuracy and inference latency (Yang et al., 2021). Burrello et al., 2022 derives TCNs by the use of MorphNet, a NAS tool which they enhanced to focus on low latency and energy consumption architectures. The use of grid search for hyperparameter tuning is seen throughout several papers (Daghero et al., 2022, Sakib et al., 2020, Coelho et al., 2021). Another study adopts exhaustive search to find the best configuration (Tesfai et al., 2022). Current research points however to NAS still being in its infancy and to its high computational requirements (Shuvo et al., 2022).

Architectural design is also linked to hyperparameter tuning, however it is an optimization we relate to improved or augmented architectures and modules that increase performance and use less computational resources. For instance, Wong et al., 2022a designs and implements a binary CNN (bCNN), that not only simplifies multiplications and reduces MAC operations but its structure can reduce the number of registers and clock cycles. Binary designs thus trade MAC operations for bitwise XNORs, typically allowing for faster processing (Daghero et al., 2021). On the other side, there are at least two main disadvantages to binary ANNs: limiting network weights and activations to a 1-bit precision can lead to unnecessarily large architectures and accuracy drops (Daghero et al., 2022). Other architectural choices include implementing ESNs instead of RNNs due to better memory and computational time (Cerina et al., 2020), or LSTMs over CNNs for video processing models (S. Li et al., 2022) and over MLPs (Chauhan et al., 2018), GRU instead of LSTM (Faraone and Delgado-Gonzalo, 2020), using depthwise convolutional operations instead of traditional ones for CNNs (Yang et al., 2021, Tesfai et al., 2022, Ay et al., 2022, Kalgaonkar and El-Sharkawy, 2021, S. Zhao et al., 2021) and the use of group convolutions (Tesfai et al., 2022, Qian et al., 2022). Other architectural choices include function-merging and block-reuse technique to reduce

registers and clock cycles (Wong et al., 2022a), the use of attention modules (S. Liu et al., 2020), linear bottleneck blocks (S. Zhao et al., 2021), and pyramidal network design using less hardware requirements and leading to shallow models (Peluso et al., 2021).

Lastly, there are other optimizations that can be applied at training-time. The paper by Coelho Jr et al., 2021 shows how streamlining layer-wise quantization during training can reduce model size and energy use while effectively maintaining accuracy. Training-aware quantization is in fact popular and appears in many of the considered papers, but we observe this happens mostly for CNNs (Van Nguyen et al., 2021, Qian et al., 2022, Burrello et al., 2022, Daghero et al., 2022, Chandna et al., 2023, Schneider et al., 2020).

4.2.4.3 Inference-time Optimizations

Increased feasibility for deployment and inference on resource-constricted devices can be achieved by optimizing models after they have been trained. Most recent developments aimed at providing efficient ML deployment are essentially compression techniques that help us limit the memory usage and computation requirements by reducing the memory requirement for storage and computation costs respectively. According to Lalapura et al., 2021, compression follows a two-step process: first by reducing the redundancy in the network, and secondly by reducing the redundancy in the bits representing the network. A disadvantage to compressing methods is that they are lossy, sacrificing performance (Coelho Jr et al., 2021).

The most popular compression method we have seen in our review is quantization. Quantization is a promising optimization step in achieving lightweight versions of pretrained models. Quantization reduces the sizes of the learned parameters (i.e weights, activations) to a lower floating point precision or a fixed-point precision. In the survey by Shuvo et al., 2022, quantization is said to have three benefits: memory saving due to the low-bit representation, reduced complexity of arithmetical operations which results in reduces latency, and improved energy efficiency. It is possible to convert learnable parameters into the same low-bit representation in all layers, filters, channels, weights and acti-

vations, or apply a mixed-precision quantization strategy, if the hardware supports this. The quantization degree varies and can even be sub-byte, for example binary (weights constrained to -1,+1) or ternary (weights constrained to -1,0,+1), turning the networks into what we call binarized/ternary networks, which may come at high performance expense. A clear limitation is that their performance is acceptable only for small datasets (Shuvo et al., 2022). A combination of both sub-byte and mixed-precision quantization can help find optimal accuracy-memory architectures (Daghero et al., 2022). Quantization enables deployment onto low-power resource-constricted devices such as Arduino Uno (Kumar et al., 2020). We see that one popular quantization choice is quantizing learnable parameters to 8-bit precision (Chauhan et al., 2018, Zebin et al., 2019, Ukil et al., 2021, Bian et al., 2022, Wong et al., 2022b), however other variations are also considered, such as 16-bit floating point (Thomson et al., 2020), or sub-byte (S. Zhao et al., 2021). Although quantization leads to small degradation in accuracy, studies like Ribeiro et al., 2022 find improved inference time, and a significant reduction in model size. 8-bit quantization is not only efficient as it can enable a reduction by four times the memory footprint from the 32-bit representation, but for some low-power hardware devices might even be necessary in order to allow parallel computations via 2-way SIMD (Peluso et al., 2021). Some hardware might not have a floating-point processing unit, so fixed-point implementations are necessary, which also turn out to be around 15% times faster than their floating-point counterparts Wang et al., 2020. Challenges that follow with the use of quantization are increased possibility for information loss due to low bit width representation, distorted network architecture, and complicated differentiation in backpropagation (Shuvo et al., 2022).

Another popular method for compression especially efficient in over-parametrized models is pruning. Pruning strategies aim to identify and remove minimal-impacting learnable parameters (e.g. by setting values to zero). This means removing redundant nodes and/or connections. A pruned model is typically retrained in order to ensure that remaining weights are still able to capture the pattern in data. In D. Liu

et al., 2022 two branches of pruning are exemplified: structured and unstructured. Unstructured pruning removes the weights with magnitude lower than a given threshold, and their corresponding connections, but it is done in a irregular manner, increasing sparsity (sparsity is the ratio between zero and non-zero parameters). Structure pruning sets whole groups of weights to zero, removing actual structures from the model, thus being able to accelerate inference. Pruning is typically done iteratively, using a magnitude-based approach where importance of the weights or nodes are determined by their magnitude or absolute value. Magnitude-based pruning is used in Dey and Roy, 2020, where pruning hyperparameters like sparsity and scheduling are also further explained. By using various sparsity settings S. Zhao et al., 2021 achieves energy-efficient architectures of only few kilobytes. The work in Yu et al., 2022 focuses on using iterative unstructured pruning on top of a model which has been subjected to structured pruning, reducing the number of parameters even further. A pruning method using dropout-based sensitivity analysis is used in Bahrebar et al., 2021. Structured weight pruning is employed in Risso et al., 2022 in order to find optimal TCNs. When it comes to LSTMs, Xiong et al., 2020 shows how sparse weight matrices resulting from pruning can be followed by a cluster-based weight sharing strategy, reducing computations and storage needs. Pruning is considered orthogonal to quantization (Peluso et al., 2021), being often used together, in which case we talk about "joint compression" (Shuvo et al., 2022). S. Zhao et al., 2021 compresses NAS-found CNNs by quantization to various bit lengths (i.e. 16,8,4,2,1 bits) and applies pruning as well. Two advantages of pruning are minimizing the physical size of parameters and reducing the inference time (Shuvo et al., 2022). While in shallow networks, weight pruning might compress weights with negligible accuracy loss, however in deeper networks, it may introduce significant loss of accuracy (Shuvo et al., 2022).

Singular value decomposition (SVD) is a compression technique where the weight matrix is decomposed into three blocks: left and right singular vectors and a diagonal matrix (Lalapura et al., 2021). The diagonal elements are a good representation of the original matrix, thus reducing

the memory required to store the whole matrix (Chauhan et al., 2018). Tensor decomposition is generalized to high-dimensional tensors. Tensor decomposition is used in various video-processing works (Y. Cheng, Li, et al., 2020, S. Li et al., 2022, Y. Cheng, Huang, et al., 2020).

Another optimization is knowledge distillation. Shuvo et al., 2022 defines it as a method that transfers learned knowledge from a larger teacher model to a smaller, shallow student model, which mimics its behavior. This uses soft-targets produced by the teacher model to as ground truth to training the student model. This technique is used in J. Ni et al., 2022 to build a student model with smaller computational overhead and improved performance in the context of human activity recognition. Knowledge distillation on fully CNNs is further investigated for time-series classification in Ay et al., 2022, showing that distillation on smaller models can be beneficial in most cases. The work in Ukil et al., 2021 creates a shallow student using piecewise linearly approximation, which given a input trained model and training data obtains a compressed model suitable for deployment on edge. Bohra et al., 2021 highlights benefits to employing knowledge distillation for both LSTMs, CNN-LSTMs and one-dimensional CNNs, such as maintaining accuracy and the high compression rate. This technique can also be used together with quantization, as seen in Faraone et al., 2021 and Peluso et al., 2021.

Lastly, we have some techniques which occur less in literature, such as the use of a two-level classifiers, where a lighter model checks whether or not if should activate the bigger one (Coelho et al., 2021). Early exit is another technique for deeper models, where predictions happen using only the early layers of a network. In Zouridakis and Dinakarrao, 2022 a performance-aware algorithm that determines the exit dynamically during runtime is implemented.

A very last remark for all of the optimizations above is that many papers adopt Tensorflow as backbone in their implementation. One explanation is that the selection is driven by the availability of Tensorflow Lite (TF-Lite), which allows authors to more easily implement optimizations that lead to deployment of neural networks on low-power devices

(e.g. Nizam et al., 2022, Zebin et al., 2019, Ukil et al., 2021, Bian et al., 2022, De Vita et al., 2020, Bohra et al., 2021). Few papers explore several platforms (an example being Y. Cheng, Li, et al., 2020).

To summarize the findings above, we see that optimizations can be applied both at the hardware and at the software level. It is worth noting that these categories are not mutually exclusive and most strategies involve matching several together, with respect to the use case and what is permitted on the specific hardware. We have seen both advantages and disadvantages to these approaches. While all optimizations align with our thesis goal of improving or preserving edge resources, the actual benefits are observed only when these methods are tuned accordingly, using mostly trial and error strategies at several steps in the implementation. We also note that even widely-used optimizations such as quantization and pruning are lossy, and come with the risk of losing information or vital connections in the trained model. The downstream consequence is a smaller or bigger drop in model accuracy.

4.3 Related Works and Research Gaps

Before concluding our literature review we initiate a discussion on observed works that have a similar approach to ours, their findings and encountered challenges. We will also list the research gaps existent in current literature.

While many demonstrate the effectiveness of using deep learning algorithms across a broad spectrum of applications, few papers focus on deploying and running these models on actual edge devices, mostly because of prohibitive characteristics (i.e battery, memory, latency requirements). This literature review has only regarded papers that attempt this. By doing so we are more likely to find contributions that have a similar goal to ours, but also determine whether or not there are areas of research which require more attention.

In a review by Shuvo et al., 2022 the authors present a more complete picture of realizing DL inference on edge, and enumerate common neural network design optimizations, such as NAS and compact net-

work design for CNNs and RNNs. This paper explains a large array of model compression techniques such as pruning, quantization, joint compression, KD and others. They find that there is a significant research gap in developing tools that can automatically map ANNs to hardware and assessing edge inference performance through benchmarks. However hands-on optimization implementation and inter-optimization relationships are not explored. There are also other surveys included in this review, such as Lalapura et al., 2021 who focuses on RNN compression techniques at edge, and the work by Shafique et al., 2018 which is focused on current design and implementation challenges for ML on IoT devices, but the same conclusions can be drawn. A more practical example by Yao et al., 2019 shows the steps to creating a execution time profiling tree based on many runs of different operations (pertaining to CNNs, RNNs and MLPs) and different devices, which holds information that can be used to compress input models based on the ones seen previously. However their work tackles inference latency only. Another contribution by Bohra et al., 2021 has a more generalized approach, performing measurements of accuracy, memory and time utilization on both traditional ML algorithms and classical DNNs on a publicly available dataset. However, their approach only considers KD as a compression mechanism, and does not explore relations between various model sizes.

All in all, when designing and optimizing a model architecture for a given application, one needs a good understanding of the relationship between the accuracy, memory, power, energy and time resources. While conducting the literature review we show that there has not been a generalizing approach that captures optimization performance across several types of ANNs and across model complexities. We observe that most papers focus either on one architecture only or one architecture at a time within the same study. On another note, the attention is typically directed towards performance improvement for a given task, while resource optimization is only performed as a minimum necessity for deployment or neglected entirely. There are only a few papers that make use of more than one inference-time optimization, so the poten-

tial of this approach is not fully explored. Only a few papers have provided benchmarks for comparison between non-optimized models and their optimized counterparts. We may thus state that the adoption of resource-aware optimization techniques is in its early stages and requires more attention, especially from a more general point of view. Consequently, a clear research gap emerges as current literature focuses mostly on specialized architecture implementations but fail to discuss general ways of improving deployment and inference on edge.

4.4 Thesis Contributions

We communicate our contributions to this area of study. We want to provide an answer to whether it is possible to find a optimal set of optimizations that can be applied across architectural and complexity dimensions, with the goal of achieving practical models.

At this point, we are able to make a distinction between "one for most" and "one for one" strategies. What has often been encountered in this literature review is the so-called "one for one" approach, that is, optimizing a model for a specific target and scenario. While this is suitable for that particular setting, it suffers from poor scalability since it does not guarantee good performance in other scenarios (Lu et al., 2019). Nonetheless, a "one for most" strategy is also lacking in the following sense: it is difficult to address model and edge device heterogeneity, or the large diversity of use case scenarios. While we are indeed limited under these terms, our effort in providing a more generalized approach is grounded in executing a throughout and multifaceted literature review that reflects common use of techniques. This approach narrows the exploration path to those optimizations that are generally applicable over a series of model architectures. This is how this process guides our PoC design and implementation choices towards a framework that is aimed for more general use. We argue that many works could benefit from eliminating tedious tuning and compression processes by using the knowledge gained from this thesis. Proving our thesis statement helps build a bridge over the gaps highlighted in the previous subsection. We

also enlarge the stock of knowledge around supporting industrial edge intelligence by using deep learning-based techniques fused with sensor technology. This thesis brings a five-fold contribution as follows:

- We perform a detailed literature review which captures novel and general trends used to enable and perform edge intelligence.
- We implement both MLPs, CNNs and LSTMs for a real-world application scenario that processes time-series data from a series of wearable devices and simulate their deployment, varying not only over model architectures but also model complexities.
- We compress our models using common inference-time optimizations, varying over their settings and combining them together.
- We construct a benchmark for comparing full-sized models and their optimized counterparts, by defining a set of performance and resource-relevant KPIs, and identifying relevant relationships between these.
- We search for a general, optimal set of resource-focused ML optimizations that can be used across architectures, by capturing the trade-off and relationship between accuracy and resource efficiency for various optimization techniques applied at multiple levels and choosing those that minimize both resource use and performance loss.

The next section walks through the actual implementation of the PoC, where design choices are based on the findings from this section.

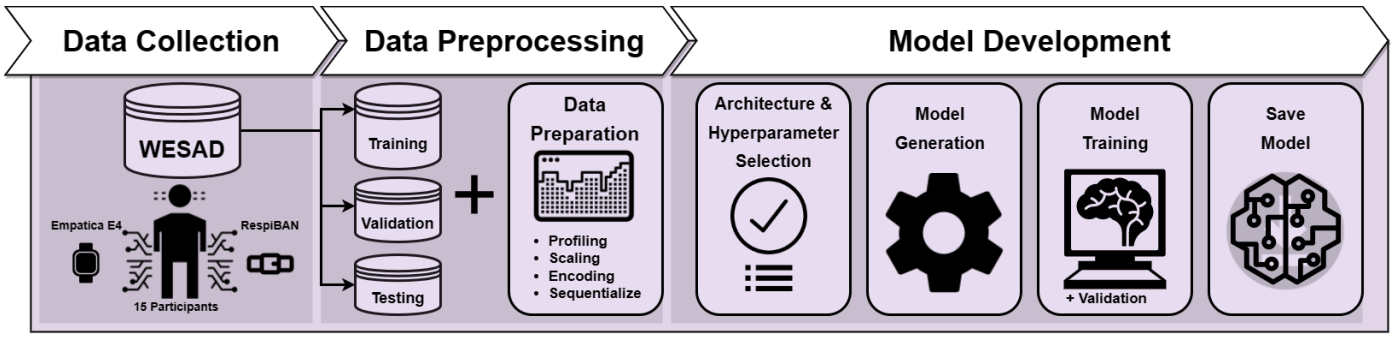


Figure 8: Model development workflow.

5 Proposed Approach

The PoC development process entails a multifaceted approach, where we not only have to consider model development under the bounds of our use case, but also development of the framework for finding a generalized set of optimizations. We can now connect the dots from the previous sections, and offer a brief description of the steps necessary to build our framework. For a full PoC characterization, we recommend the reader to return to the overarching thesis statement and success criteria from section 2. We will separate our framework building activities into three phases: the model development phase, the model optimization phase and lastly, the model deployment and inference phase. In this section we will however only discuss the implementation, and not the actual interpretation, which is a topic reserved for later sections.

5.1 Model Development

The process commences with model development, that is, constructing the full-sized models following what one might consider a typical ML recipe: data collection, data preprocessing, architecture and hyperparameter selection, model generation, fitting the models to the data and finally, saving the models to a proper format, prior to further optimization and deployment. Figure 8 represents the entire model development process workflow.

5.1.1 Data Collection

We begin with presenting, profiling and processing a time-series data set, a necessary step prior to fitting models. Here, time-series data preparation techniques will be harnessed.

Our data collection journey has essentially started with a data set search procedure, in which several candidate data sets have been considered (see appendix A). Having the possibility of using a ready-to-load data set evades the need to invest time and other resources in collecting data, but it also trades at a cost. Consequently, we pay in increased cautiousness around data quality and assuring that the collection process adheres to standards, principles and good scientific practices. While transparency and trustworthiness are taken for granted, the best option at hand will be to explore the data and see whether or not we observe any discrepancies. Our initial problem analysis had envisioned a task using wearable sensor data as the main use case for our tool, and so relevant data sets were suggested in this direction.

After comparing properties of candidate data sets, we eventually decided to use *Wearable Stress and Affect Detection* (WESAD), an open-access data set intended for wearable stress and affect detection. Regarding the choice of the data set, it was driven by three requirements. The first requirement is choosing a time-series data set. This choice allows us to employ various architectures, including time-series specialized ones. As seen in section 3.2, this is also a very popular type of data, seen in many industrial applications. The second requirement involves the label type. We are interested in doing binary and multi-class classification, and so we want to have discretely labelled instances to enable supervised learning. Third and last, we wish to have a data set of good quality and size. Fulfilling this requirement allows the models to train on representative data and avoid overfitting.

To collect the WESAD data, a laboratory study was performed where affective stimuli were used to elicit stress and amusement states in 15 participants (12 males, 3 females, age in years: 27.5 ± 2.4) by the use of two devices: one wrist-worn device (Empatica E4) and one chest-worn

device (RespiBAN Professional). The RespiBAN sensors capture acceleration (ACC), electrocardiogram (ECG), electrodermal activity (EDA), electromyogram (EMG), and temperature (TEMP) information at 700 Hz sampling frequency, while Empatica E4 records blood volume pulse (BVP), ACC, TEMP, and EDA, at the following sampling frequencies: 64 Hz, 32 Hz, 4 Hz and 4 Hz respectively. Further details on the process of data collection can be found in the introduction paper on WESAD (Schmidt et al., 2018), where an overview over engineered features, a benchmark using traditional ML models and other details are listed. Participants were subjected to four different states: sitting at a table reading magazines which constitutes the *baseline*, an *amusement* condition by showing funny video clips, a *stress* condition where they had to participate in a public speaking and do a mental arithmetic task, and a "de-excite" period called *meditation* where breathing exercises were performed. The study had an approximative timeline of two hours. In the actual data set, all conditions are identified by their corresponding labels: 1 for baseline, 2 for stress, 3 for amusement, and 4 for meditation, while 0,5,6,7 were ignored as they are not defined. The WESAD provides a synchronized data file for each participant, where the raw data for all devices are already aligned.

WESAD is thus a multimodal data set that has been collected in such a way that makes it a collection of time-series data, fulfilling our first requirement. Both labels and self-reports are provided for each instance in the data set, fulfilling the second requirement. This makes this data set not only suitable for stress-prediction tasks, that is, binary classification where all instances are labelled either as stress or non-stress, but also suited for multi-class classification task between stress, amusement and neutral states. Self-reports add another dimension to what this data can be used for (e.g. personalized predictions, assessing positive and negative affect states, anxiety levels and type of stress), but we will use manually defined labels, while the rest goes beyond the scope of our thesis. Some of the data has been discarded due to sensor malfunction, resulting thus in a cleaner data set. There is also a separate file that includes the synchronised data from both the

RespiBAN and the Empatica E4 devices, which simplifies the data processing phase. While the raw data set size is reasonable, we will see that adjusting the window size and the overlapping area between windows of data can increase or decrease its size, adding more flexibility. All three requirements have been fulfilled by this point.

The benchmark in Schmidt et al., 2018 achieves up to 93.12% accuracy in the binary classification task (stress vs non-stress) when trained on a linear model using RespiBAN features. It should also be noted that recent studies using ANNs on the WESAD data achieve even higher accuracy and unlock new insights about the data. A CNN-based approach where raw time-series are encoded into images achieves an even higher accuracy of 94.77% in a multi-class stress level prediction task, using purely features related to the chest (Ghosh et al., 2022). Another deep ANN generated by NAS manages to achieve an accuracy of 93.14% based only on recordings from the Empatica E4 wristband (Huynh et al., 2021). Lightweight deep neural networks trained on the WESAD data set also achieve results that are comparable or better than state-of-the-art approaches (Chatterjee et al., 2022).

This is indeed a motivating factor to explore the use of deep learning architectures on time-series data and reap the benefits that come with this technology. However, our framework is not built with the intention of maximizing prediction power, but rather concerned with preserving performance while lowering resource utilization.

5.1.2 Data Preprocessing

In this subsection we will go through our design choices and assumptions made for the data preprocessing step.

As a first step we load the raw WESAD data, fetching both Respiban and Empatica variables (total of 14 variables). For simplicity, we will stick to using raw data values in our implementation, whilst agreeing that feature engineering can indeed be a powerful step in reducing input data complexity. However, as we have explained in 3.1, neural networks are typically good candidates at processing raw data. Due to

the fact that these two devices use different sampling frequencies, we choose to do a linear interpolation of the values not covered by the original sampling, removing non-numeric values and keeping the highest sampling frequency of 700 Hz as a reference. In this case, 700 lines of data are equivalent to one second of recording. In this initial phase we also set the labels for both binary and multi-class tasks. For the binary case we combine the non-stress states and into one state (i.e 0), while the instances belonging to the stress state are labelled as such (i.e 1). A similar approach is done for the multi-class case: here instances belonging to stress and amusement conditions get their own labels (i.e 1 and 2), while all other instances are combined into one state (labelled as 0).

The next step involves splitting the data into training, validation and test data. The training set is used for the actual learning process, the validation set will be used as a stopping criteria while training to avoid overfitting the model, and the test set will be hold out for some intermediary testing and for the inference data. Prior to splitting the data, we randomize the order of the participants to remove any biases. We decide to keep data from two participants for the test set, while the rest of the data is split into a 80-20% proportion for training and validation, respectively. Now what we have three separate data sets, we move on to scaling the data.

According to Chatterjee et al., 2022 normalization helps reduce the inter-subject variation and suppresses noise. Two scalers were tested: the min-max and the standard scaler. The min-max scaler converted all values to a range between 0 and 1, while the standard one mapped all values to a normal distribution, allowing negative values as well. The scalers were first initiated on the training data and then the rest of the data was transformed using the same scaler. In practice we observed that the standard scaler resulted in less overfitting for some model architectures, which is why we chose it for the data processing.

Then we moved to creating sequences of data, which represent the inputs to our models. We will use a sliding window technique, typically used in sequencing time-series. According to Wei and Radu, 2019 there are two main reasons to use samples overlapping. The first is to en-

hance dependency between consecutive instances by exposing repeated information in the overlapping parts, and secondly, a higher overlap allows us to generates more unique instances for training. Especially for recurrent models, this has a strengthening effect through memory adjacent actions and features over consecutive inputs. For the CNN, overlapping windowed data helps convolutional filters explore temporal relationships between the samples within an activity. Our choice for the most optimal window size was done in an empirical manner, trying out several window sizes. While sequencing we also make a choice as to how each window will be labelled. For the binary task, we choose to label the whole sequence as stress if at least one instance is labelled as such, while for the multi-class case we choose the label that occurs more often. We then encode these labels to their one-hot representation. By picking a window size of 3 seconds (or 2100 lines) with an overlapping area of 50%, the sequencing results in a training data set containing 19953/19951 instances, a validation data set of 6011/6017 instances and a test set of 4004/3999 instances for binary and respectively multi-class tasks. The small variation in number of instances comes from the participant randomization prior to splitting data sets for both tasks.

In a study by Ghosh et al., 2022, it is stated that ML algorithms perform better when numerical input variables and output variables have a standard probability distribution. To verify this and whether or not our data contains outliers or noise and/or is skewed, we run some profiling on the data. We use the ydata-profiling package to report unwindowed data statistics (all reports are included in our repository referenced at the end of this section). By inspecting the data this way, we made sure that there are no missing values. We also find that the data is imbalanced for both binary and multi-class cases (approximately 77.8% non-stress data for the binary task, while multi-class has 65.3% non-stress instances and only 12.4% amusement data). Most features follow an approximately Gaussian distribution (see figure 9 for Respiban features, and figure 10 for Empatica features), although others are clearly skewed (e.g. chest and wrist EDA). Some outliers are present, however using the IQR elimination rule resulted in a significant reduction in

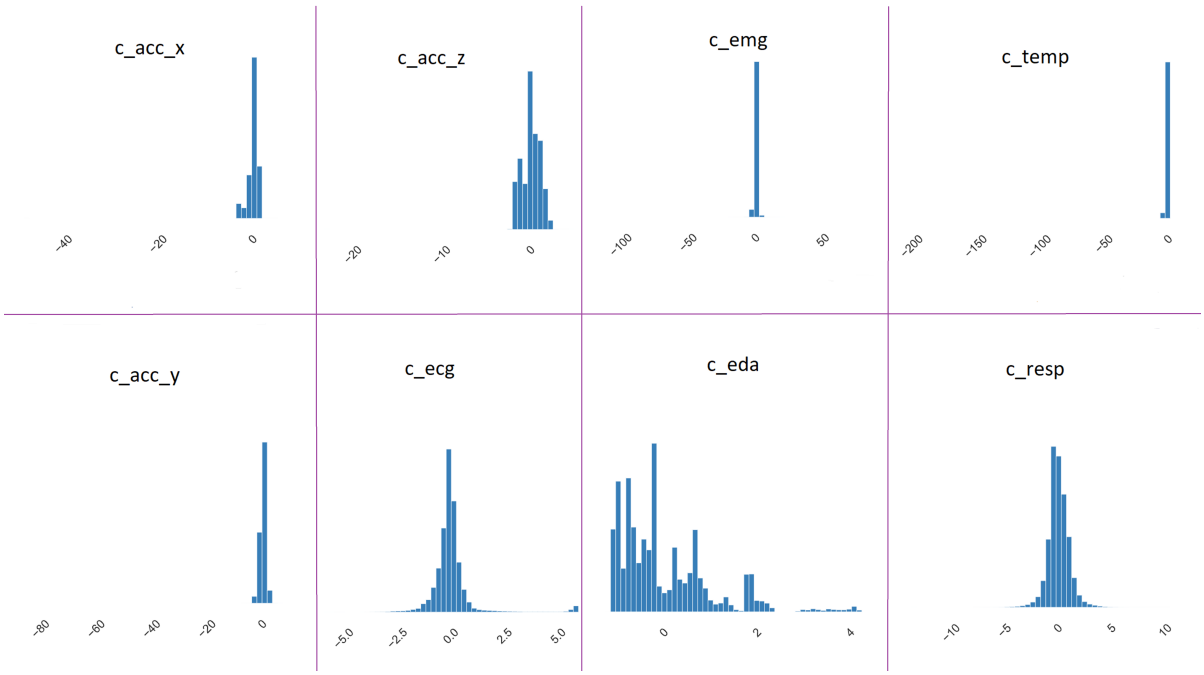


Figure 9: Chest-device variables distributions (binary train set).

non-neutral labelled data, so we decided to keep all instances. We have also computed heatmaps over feature correlations, however the train set was too large to enable this function, so we have created a heatmap separately for the first half of its content (see figures 11, 12, 13). We observe that some correlations are more accentuated in the validation and test set, however this is normal given that fewer participants are considered. We also observe more negatively correlated variables in the training and test test than in the validation set but the general pattern is still maintained. An explanation to this is the fact that the number of participants is quite small, so differences between participants are more accentuated when profiled.

5.1.3 Model Generation

We continue the development with model generation, which in short is the step where we create various models of different complexities and architectures. By answering RQ1, we argue that considering and implementing variations over CNNs, LSTMs and MLPs is sufficient to

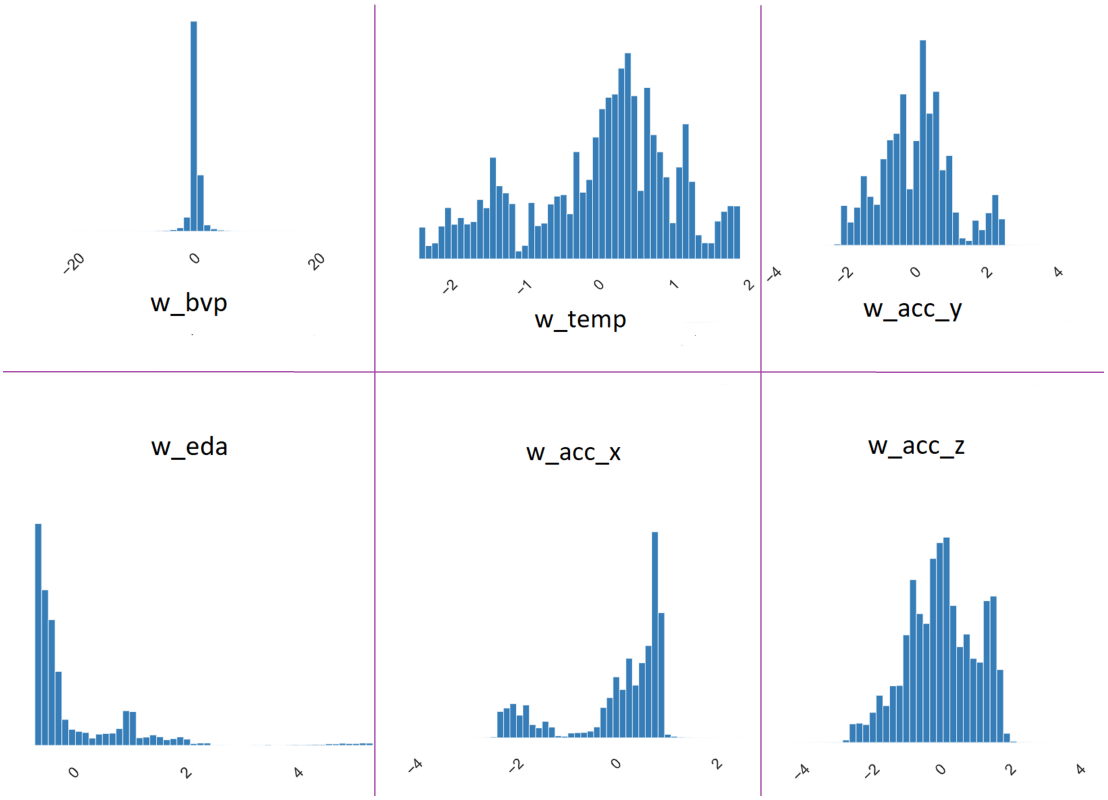


Figure 10: Wrist-device variables distributions (binary train set).

provide knowledge around the most common architectures. These architectures have also been presented in section 3.1. We also remind that some models perform better for specific problems, which is better portrayed in coming evaluations. We might not find a very accurate model, but this is also not the goal of our study, so there is no need for an extensive search after the best performing model.

Now that we have the preprocessed dataset and a set of architectures in mind, we want to choose hyperparameters that result in similar model complexities. Hyperparameters are configuration parameters that define the model and control the training process. They are specified before training the model and eventually tuned, which is where the validation set comes into the picture. Both architectural choices and hyperparameter selection are vital in addressing issues such as vanishing/exploding gradients, overfitting and escaping local minima.

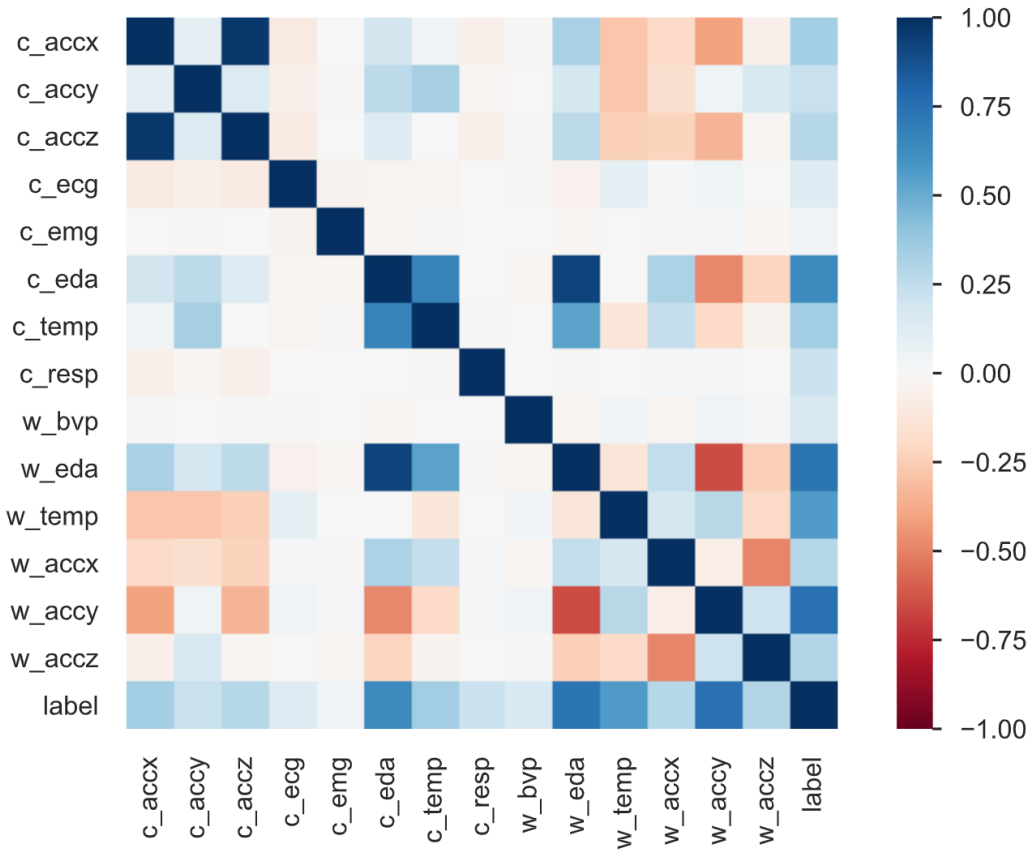


Figure 11: Binary train data heatmap.

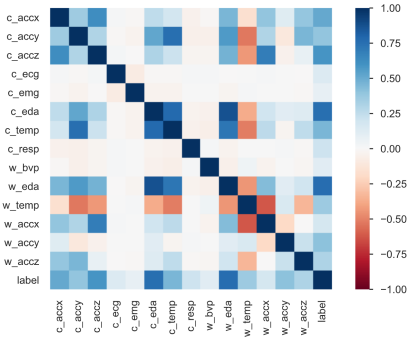


Figure 12: Binary validation data heatmap.

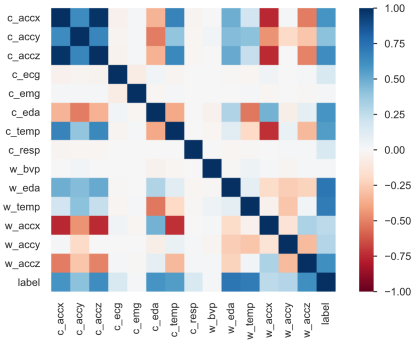


Figure 13: Binary test data heatmap.

We use the number of learnable parameters as reference to model complexity/size, based on the RQ3 answer from section 4.2. We will consider six model sizes, from thousands of parameters up to over a million. These ranges are chosen as follows: 50-100k, 200-300k, 400-500k, 600-700k, 800-900k and lastly 1-1.1M. This selection is based on findings from literature, where we have seen models ranging from a couple thousand parameters to very large ones, but also it was our intention to have models where the full-size versions can be deployed to the device of choice, in order to enable our comparative analysis. We thus generate one model within each one of these ranges, for each model architecture, covering a wide array of sizes. Notation-wise, we will use a underscore for model names to refer to their sizes: for example MLP_1 reads as the MLP with a number of parameters belonging to the first range. Table 2 goes over the actual parameter count for each model for the binary task (the multi-class task has a tiny increase in parameters depending on how many units are found in the last layers, but remains still within the range interval).

Size	Parameter range	MLP	CNN	LSTM
1	50,000-100,000	58,805	61,601	73,345
2	200,000-300,000	235,217	277,761	235,713
3	400,000-500,000	470,561	497,921	474,753
4	600,000-700,000	647,479	678273	606,337
5	800,000-900,000	884,111	862,721	885,057
6	1,000,000-1,100,000	1,061,765	1,049,409	1,003,905

Table 2: Number of parameters present in each architecture.

The architecture is different for each model, but there are some common design choices taken along the way. We varied the number of layers in the proposed models from 1 to 4, contributing to the depth (for the CNN this is without the fully-connected module). For all models, a 0.5 drop-out rate has been added as a regularization method between various layers. The hidden activation is ReLU for all models apart from the LSTM which uses tanh, and the output activation is the same (sigmoid for binary, softmax for multi-class). Crossentropy loss is used as a loss function for both tasks. We adopt adaptive learning rates by the

use of the “adam” optimizer, with the default learning rate of 0.001. There are also some model-specific hyperparameters: for the CNN this is reflected by the number of filters, kernel sizes, and pooling layer specifications. The kernel size allows capturing temporal correspondence between neighbouring data points within the filter, so we should keep them big enough in the first layers as to not lose temporal relationships. We choose to always use a global pooling layer after the last convolutional layer, as we have seen benefits to this approach (S. Zhao et al., 2021, Faraone and Delgado-Gonzalo, 2020). We also use max pooling for some of the previous convolutional layers. For all the CNNs we use the same flatten layer plus a two-layered fully-connected module (each followed by drop-out and displaying 64 and 32 nodes respectively) to convert feature maps into actual outputs. A fuller overview of hyperparameter and architectural choices can be found in table 3, table 4 and table 5. These architectures are shared for both binary and multi-class tasks. All models are implemented using Tensorflow.

Table 3: MLP architectural layouts.

Size	Number layers	Number units	Layer followed by dropout
1	1	2	1
2	1	8	1
3	2	16-8	2
4	3	22-16-16	1-3
5	3	30-32-32	1-3
6	4	36-32-32-32	2-4

Table 4: LSTM architectural layouts.

Size	Number layers	Number units	Layer followed by dropout
1	1	128	1
2	1	256	1
3	2	256-128	2
4	3	256-128-128	1-3
5	3	256-256-64	1-3
6	4	256-256-128-64	2-4

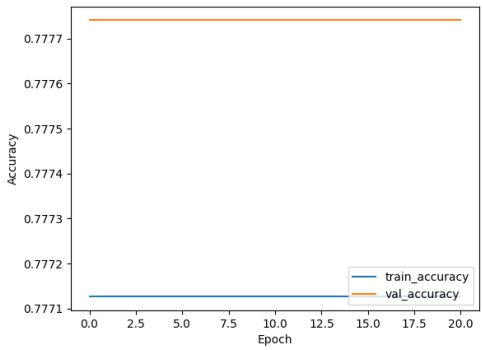
Table 5: CNN architectural layouts.

Size	Number Conv layers	Number filters	Kernel size	Pooling type	Layer followed by pooling	Pooling size
1	1	32	128	Global max	1	-
2	1	64	256	Global max	1	-
3	2	64-64	256-64	Global max	2	-
4	3	128-64-64	64-64-8	Global max	3	-
5	3	128-128-128	256-16-8	Max, global max	1-2-3	64-1
6	4	128-32-64-32	256-128-4-4	Max, global max	2-4	64

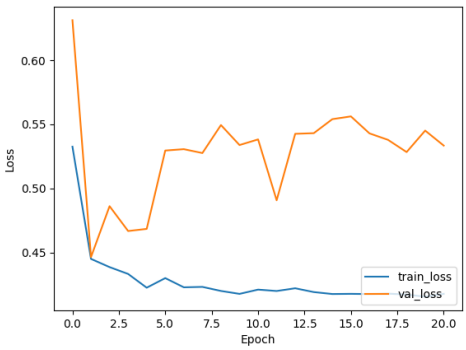
5.1.4 Model Training

During training, the model goal is to optimize learnable parameters so that predicted labels match the true labels as much as possible, using a loss function to measure the difference between predictions and true labels. Before training we set a termination criteria so that we do not end up overfitting the models. This is also where the validation set comes in, which is fed into a early stopping callback where a patience of 20 epochs has been set that monitors changes in validation accuracy. All models train a minimum of 20 epochs, and a maximum of 100, using a batch size of 64 samples, apart from the LSTM which always stops training after maximum 25 epochs due to its long training time. While larger batch sizes are computationally efficient, smaller batch sizes lead to better generalization. Higher numbers could of have been chosen for epochs, but we are talking about a large number of models, so epoch number was reduced to keep to the planned thesis timeline. We also reshape the input for the MLP from the windowed shape (X, 2100, 14) to (X, 2100*14), that is, X samples and 2100*14 features per sample. X depends on the task.

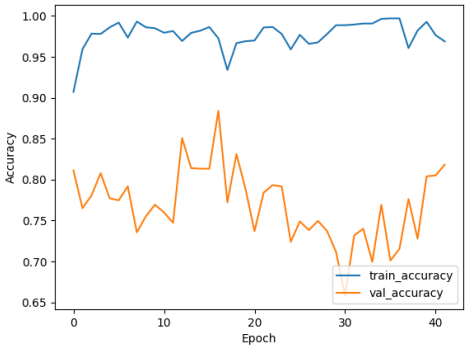
This phase results in a set of full-sized models, of varying performance. The highest accuracy found on the entire test set reaches 0.88 for the CNN_5 in the binary task and 0.675 for MLP_4 in the multi-class task. We have also plotted the accuracy and loss on both training and validation for all of the models, to further investigate the training process. We will only give an example for the best performing model for each architecture (see figure 14): MLP_1, CNN_5 and $LSTM_5$, however all other plots can be found in the repository. For MLP_1, we observe



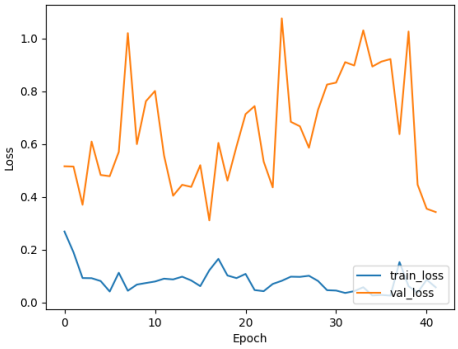
(a) MLP_1 training accuracy.



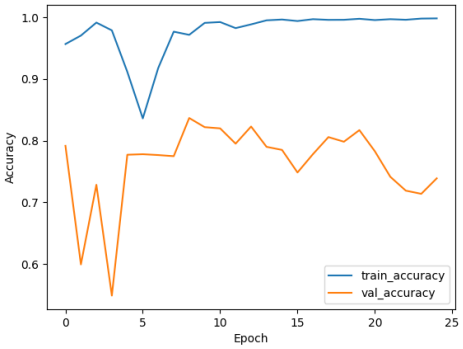
(b) MLP_1 training loss.



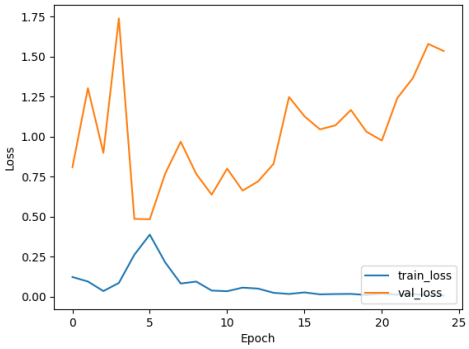
(c) CNN_5 training accuracy.



(d) CNN_5 training loss.



(e) $LSTM_5$ accuracy plot.



(f) $LSTM_5$ training loss.

Figure 14: Accuracy and loss for best performance models at training.

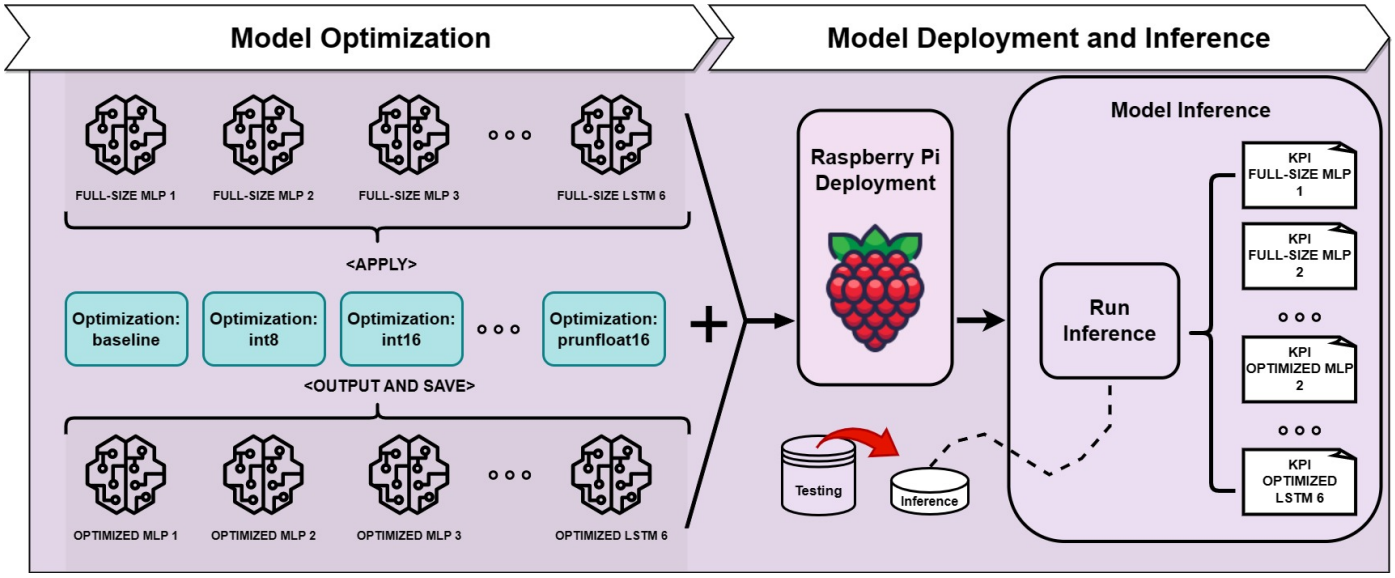


Figure 15: Model optimization, deployment and inference phases.

that even if the accuracy is quite high (i.e 0.777), the model is stuck into a local minima and cannot escape. We saw that its recall and precision (and implicitly F-score) are in fact zero. This is because it chooses to label everything as a non-stress value. It also seems that CNN_5 has been luckily initialized, since its accuracy is high starting already from the first epoch. $LSTM_5$ does not have the best accuracy overall, but there is at least variation between epochs, and one could argue that training for 25 epochs might not be sufficient. All in all we see that most models have low-to-average accuracy and may not be very stable, however, as mentioned in earlier stages, our thesis goal is not achieving high-quality models, but rather bring attention to the impact various optimizations have on models of different complexities.

As a final activity, we save all of the trained full-sized models in a default Keras format. The model architecture, the trained weights and the optimizer state are thus saved, and ready to be accessed at a later time.

Table 6: Model optimization descriptions.

Optimization ID	Optimization Short Description
baseline	Dynamic range quantization
int8	Quantization integer only (8-bit)
int16	Quantization integer with float fallback (16-bit)
int32	Quantization integer with float fallback (32-bit)
float16	Quantization float (16-bit)
prun	Magnitude-based weight pruning
prunint8	Magnitude-based weight pruning + Quantization integer only (8-bit)
prunint16	Magnitude-based weight pruning + Quantization integer with float fallback (16-bit)
prunint32	Magnitude-based weight pruning + Quantization integer with float fallback (32-bit)
prunfloat16	Magnitude-based weight pruning + Quantization float (16-bit)

5.2 Model Optimization

An essential phase to our PoC is the optimization phase where individual or combinations between resource-focused optimizations are applied to the previously trained full-sized models (see figure 15). These optimizations are found by scanning the list of post-training optimizations from answering RQ4, for those optimizations that can be applied over several types of model architectures. Quantization and pruning are most prevalent in literature, and also observed being applied over various architectures. We pick these as candidates to our set of generally-applicable optimizations. Although only two types of compression techniques are considered, we will have ten different optimization variations (more details in table 6). At a minimum we aim to include the most common variations found by answering RQ4 in subsection 4.2.

Our goal in this subsection is to optimize our full-sized models, and thus for each trained full-sized model from subsection 5.1, we apply the entire array of optimizations presented in table 6. Our network count by the end of this process is 18 full-sized models, each optimized 10 times, resulting in 180 lightweight models, adding to a total number of 198 models per task (or 396 models including both binary and multi-

class tasks). We will now briefly detail the implementation of these techniques, and also explain any assumptions that have been made:

- **baseline**: A dynamic approach to quantizing a model post-training, recommended as an initial optimization step by Tensorflow. Weights are quantized post-training and activations are quantized dynamically at inference, by doing estimations based on a representative data set. The converter applies a scale factor to each weight and activation given these estimations. We have not provided this data set ourselves, however a default set of random tensors with the same shape as our inputs is used instead. The quantization aims for a fixed-point model with 8-bit integer precision. We see that all other quantization techniques build on this.

```
converter_baseline = tf.lite.TFLiteConverter.
from_keras_model(model)
converter_baseline.optimizations = [tf.lite.Optimize.
DEFAULT]
tflite_model_baseline = converter_baseline.convert()

```

Listing 1: Optimization implementation (baseline).

- **int8**: This is a full integer quantization approach, and estimating the range of all tensors, including variable ones (e.g. input, output) in the model requires a representative data set for calibration. In our implementation, $representative\_data\_gen$ stands for a generator function looping through and yielding all instances of the actual training data. This type of optimization is necessary to create compatible models for deployment on 8-bit devices such as microcontrollers.

```
converter_int8 = tf.lite.TFLiteConverter.from_keras_model
(model)
converter_int8.optimizations = [tf.lite.Optimize.DEFAULT]
converter_int8.representative_dataset =
representative_data_gen
converter_int8.target_spec.supported_ops = [tf.lite.
OpsSet.TFLITE_BUILTINS_INT8]
```

```
    converter_int8.inference_input_type = tf.uint8
    converter_int8.inference_output_type = tf.uint8
    tflite_model_int8 = converter_int8.convert()

```

Listing 2: Optimization implementation (int8).

- **int16, int32**: The quantization to 16-bit and 32-bit integer are similar to the int8 technique seen before, but we instead of quantizing all tensors to integer, we use a float fallback (meaning that the input and output tensors still use float precision). We will use the default representative dataset.

```
    # int16:
    converter_int16 = tf.lite.TFLiteConverter.
    from_keras_model(model)
    converter_int16.optimizations = [tf.lite.Optimize.DEFAULT
    ]
    converter_int16.target_spec.supported_types = [tf.int16]
    tflite_model_int16 = converter_int16.convert()
    # int32:
    converter_int32 = tf.lite.TFLiteConverter.
    from_keras_model(model)
    converter_int32.optimizations = [tf.lite.Optimize.DEFAULT
    ]
    converter_int32.target_spec.supported_types = [tf.int32]
    tflite_model_int32 = converter_int32.convert()

```

Listing 3: Optimization implementation (int16 and int32).

- **float16**: The code below shows the implementation steps for enabling float16 quantization on model weights.

```
    converter_float16 = tf.lite.TFLiteConverter.
    from_keras_model(model)
    converter_float16.optimizations = [tf.lite.Optimize.
    DEFAULT]
    converter_float16.target_spec.supported_types = [tf.
    float16]
    tflite_model_float16 = converter_float16.convert()
```

Listing 4: Optimization implementation (float16).

- **prun**: Below we will present the implementation steps for magnitude-based weight pruning. This technique iteratively eliminates model weights during the training process, starting with an initial proportion of weights (here, 40%) that are set to zero (sparsity) and increasing it using a polynomial decay function under training (here, to a maximum of 60%). Under the training process for pruning, we have chosen to use our original training data, however 20% of has been used as validation data in this process. We run 5 epochs and use a batch size of 128. The loss and optimizer is the same as the one used for training the full-size models.

```
    prune_low_magnitude = tfmot.sparsity.keras.
    prune_low_magnitude
    validation_split = 0.2 # 20% used for validation
    num_data = repr_data_X.shape[0] * (1 - validation_split)
    end_step = np.ceil(num_data / batch_size).astype(np.int32
    ) * epochs
    pruning_params = {'pruning_schedule': tfmot.sparsity.
    keras.PolynomialDecay(initial_sparsity=0.40,
    final_sparsity=0.60, begin_step=0, end_step=end_step)}
    pruned_model = prune_low_magnitude(model, **
    pruning_params)
    pruned_model.compile(optimizer='adam', loss=loss,metrics=
    metrics)
    callbacks = [tfmot.sparsity.keras.UpdatePruningStep()]
    pruned_model.fit(repr_data_X, repr_data_y, epochs=epochs,
                     validation_split=validation_split,
    batch_size=batch_size,
                     callbacks= callbacks, verbose=1)
    pruned_model = tfmot.sparsity.keras.strip_pruning(
    pruned_model)
    converter_prun = tf.lite.TFLiteConverter.from_keras_model
    (pruned_model)
    tflite_model_prun = converter_prun.convert()

```

Listing 5: Optimization implementation (prun).

- **prunint8, prunint16, prunint32, prunfloat16**: Given that these techniques imply a combination between the pruning method presented earlier and any of the quantization techniques above, the implementation of these is quite straightforward. That is, we first prune our model following the steps above and then replace the *model* variable from each quantization technique by the variable representing the pruned model (i.e. $pruned\_model$).

Note that when optimizing LSTMs, we have to disable an experimental feature and add a set of operations supported by TF-Lite for LSTMs. This code and other non-critical code bits have not been included here, to increase readability (the full implementation file can be accessed by vising the repository referenced at the end of this section).

5.3 Model Deployment and Inference

The last phase is the deployment and inference phase. In this phase we capture full-size and optimized model edge performance by deploying models to a resource-constricted device (see figure 15). By addressing RQ2, we observed that one of the more popular edge devices considered across literature is the Raspberry Pi. Because of this we will aim to deploy models to this type of device. Other motivations for this choice are also its availability and reduced market costs. We then collect the key performance indicators presented in figure 6 and figure 7, resulting from answering RQ3. This is the backbone of our framework, granting us the ability to set up a comparative analysis in the next section.

We will begin this subsection with an overall description of our experiment setup. A picture of the overall setup is found in figure 16. We aim to deploy our models to a Raspberry Pi 4 Model B device with 4GB RAM, and 64-bit quad-core Cortex-A72 processor. This device is connected to a Raspberry keyboard, mouse and a screen. Between the device power input and its power supply we attach a Uni-T USB tester. The USB tester allows us to monitor the voltage level (shortened as VDD) using volts (V) and the electric current flow (shortened as Curr) using amperes (A), among other features. On the Raspberry device we

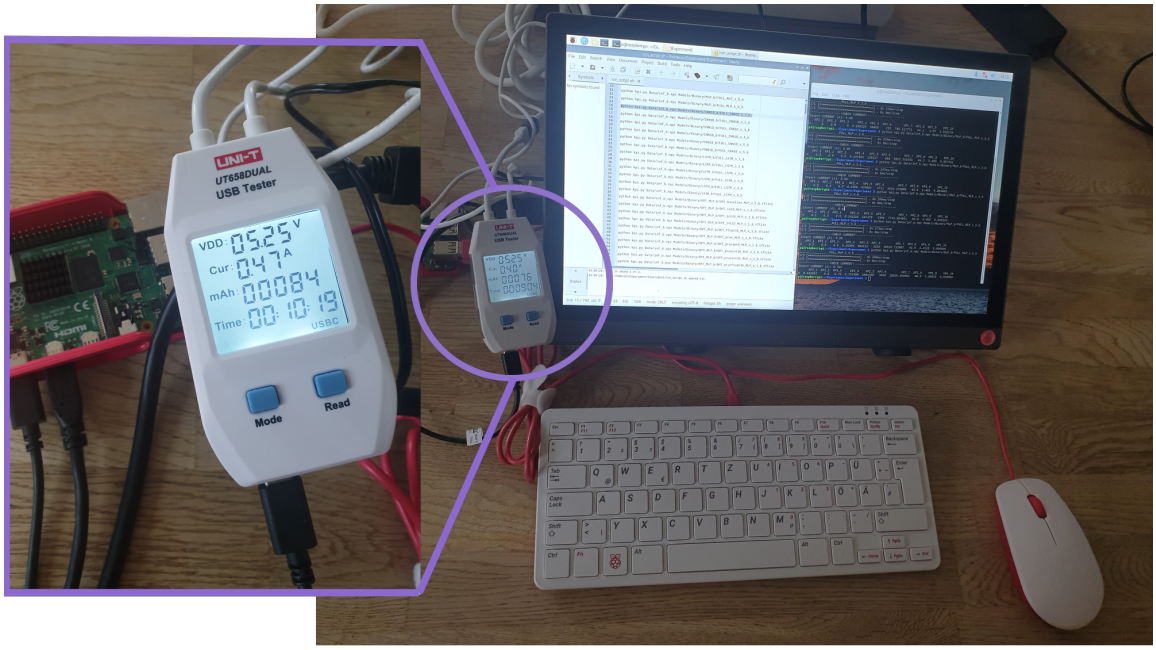


Figure 16: Raspberry Pi setup.

install the 64-bit Debian GNU/Linux 11 (Bullseye) operating system, following all necessary steps. We then install the necessary packages and dependencies for running the inference file. We send our models over to the device and store them together with the inference data, a list over executable lines, and the actual inference file that collects all metrics mentioned above. We make sure that the device is disconnected from the Internet, and that any unnecessary applications that may interfere with the model computations are closed in advance.

The inference process has been simplified so that we only need to run one command to perform inference on a given data set for a given model. Once the command is run, the inference process commences with loading the data set and the called model. When it comes to the inference data, we remind the reader about the test set that contains data from two participants. We derive our inference data from this test set, by selecting 10 (or 30 seconds) continuous windows of data for each type of label, from both its first and its last occurrence in the test data set (we do this because the data is concatenated sequentially and we want to se-

lect data from both participants). We have later used a larger set of 50 continuous windows of data to calculate accuracy-related performance metrics only, as prediction performance on 10 windows of data for each label proved to be difficult to interpret for unstable models. Note that if the called model is optimized by int8 quantization, we need to change the inference data precision to int8 as well as other representations are not allowed. We then load the model, however we note that optimized models make use of an interpreter.

The first metric we collect is the CPU usage (KPI_8), which is monitored in parallel as we are running inference on data points. We collect an array of CPU values during the inference process, remove zero values and take the average over all CPU values for the final metric. The next step is performing the actual predictions for all inference data points, and at the same time profiling this process by placing timestamps around the inference calls (KPI_4). It must be mentioned that a little time overhead might be introduced due to looping over inputs and storing predictions into numpy arrays, however this happens for both full-sized models and their optimized counterparts. Once all predictions have been stored, we can calculate the prediction performance metrics ($KPI_1 - KPI_3$). We have implemented these manually, as we tried to keep the list of unnecessary installations on the hardware device to a minimum. In the binary task, we choose 0.5 as a classification threshold, while the highest probability or value is chosen for the multi-class. Next we calculate the number of parameters (KPI_5). This is easy for the full-sized models, as Tensorflow offers a function which returns the number of learnable parameters for a given Keras model. On the other hand, for optimized models we can estimate these by using the sizes of the model tensors. Our next metric is the size of the compressed model file (KPI_6), which is collected by compressing our models and using a gzip file format. We then get the file size for this file and express it in terms of kilobytes (KB). The memory allocation on the device for the uncompressed models is also collected and expressed in terms of kilobytes (KPI_7). Power consumption (KPI_9) is measured by running continuous inference (i.e looping over inference data points) for about 10 seconds

and at the same time checking the USB tester to obtain electric current flow value (i.e Curr). The terminal then asks for a input for the current value. We observe that voltage does not fluctuate, and so we choose a permanent value of 5.25V. For the current value, we take the average of values that come on the screen, as we register that different inputs have small fluctuations in current. By obtaining KPI_9, we also are able to calculate KPI_10. Lastly, all of these metrics are written to a .csv file and saved locally, ready for further profiling.

All of phases above contribute to creating a framework that outputs an array of KPIs. With a richer picture of the inner workings of the model training, model optimization, model deployment and inference phases, we can continue to the next and most vital part of this thesis, the analysis over the collected metrics.

All models, code files, plots and reports can be accessed by visiting: https:/github.uio.no/adelaann/Artifact. Note: accessible only to UiO students and employees!

6 Experimental Results and Evaluation

This section provides a presentation of experimental results, or performance evaluations, for full-sized models alone and together with their optimized counterparts. In this section our goal is to present and shortly explain observations, where visual representations are used to essentially frame steps in the PoC approach. At the end of this section we use our success criteria to drive our choice for optimizations, constituting the evaluation phase. All of our findings from this section can be found in the Jupyter Notebook file in the repository referenced in section 5.

6.1 Full-sized Model Analysis

In order to better understand the challenges posed by deploying and running inference using full-sized models, we begin the analysis by tak-

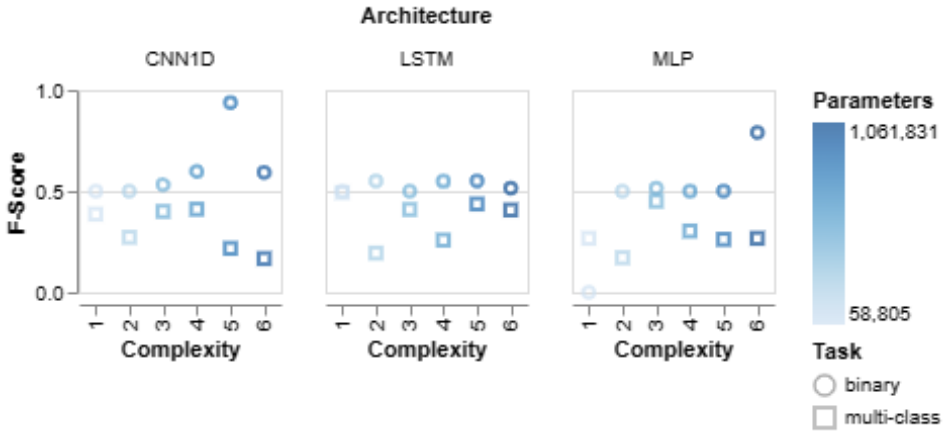


Figure 17: Full-size analysis: predictive performance versus model complexity.

ing a first look at full-sized models. First we do this through the contrastive lens of predictive performance and model complexity, as seen in figure 17. We use F-score as a measure for predictive power, as it is a balanced value between precision and recall. We see that being a larger model with more learnable parameters does not necessarily mean achieving better performance. When it comes to F-score readings, the models achieve similar results across architectures, with few exceptions, which may be due to lucky weight initializations (e.g. binary CNN_5) or models that label most data to one class (we note that MLP_6 achieves a high F-score thanks to its high recall of 0.98). Another observation is that multi-class models are consistently worse performers in terms of accurate predictions. The worst performance on inference data is seen for the binary MLP_1, and best performance for binary CNN_5. In figure 18 we note that although the number of parameters is close to each other across all model complexities, the actual memory requirements for storing the full-sized models are slightly different across architectures, with LSTMs being the most memory-hungry models.

Next, we plot the inference time in figure 19. We would like to see a roughly similar inference latency even when models become deeper and more complex. However, we can observe that this is not always the case, and model complexity can pose a constrain leading to latency, especially

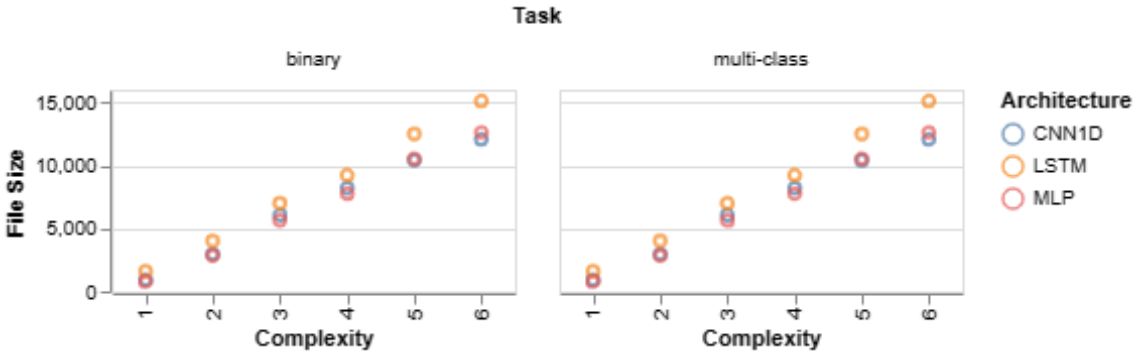


Figure 18: Full-size analysis: file size versus model complexity.

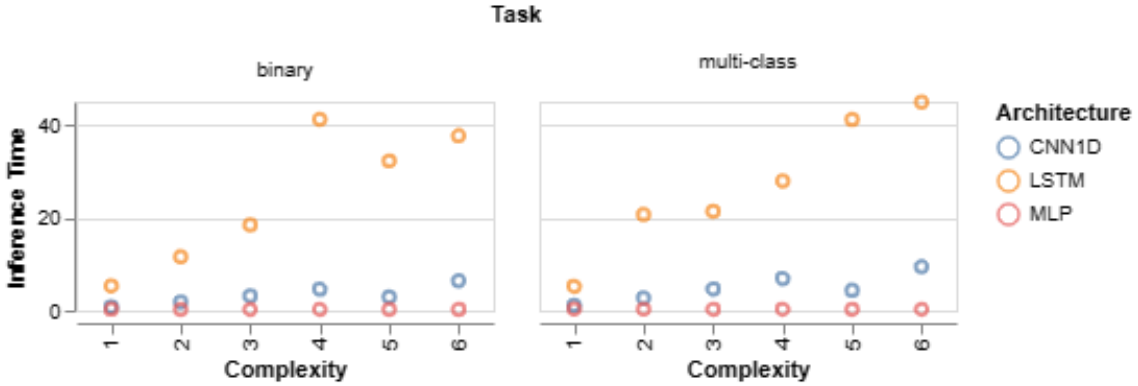


Figure 19: Full-size analysis: inference time versus model complexity.

for LSTMs whose inference time is much higher than all of the other architectures. Full-sized CNNs and MLPs use only up to a few seconds to perform inference on inference data, and CNNs use more time than MLPs. Another interesting takeaway is that even if some models are larger in terms of number of parameters, they can have lower inference time than smaller ones (e.g. binary $LSTM_4$ uses more time than both $LSTM_5$ and $LSTM_6$, and similarly for CNN_5). We will see that CPU usage can offer an explanation to this. In terms of CPU usage (see figure 20), we note that most values lay within a similar range for both binary and multi-class scenarios, using less than one CPU core, with the exception of a spike for $LSTM_6$ and for CNN_5, which appears in both tasks. These two models appear to trigger the use of multiple cores, which could be an explanation as to why CNN_5 and $LSTM_6$ have less

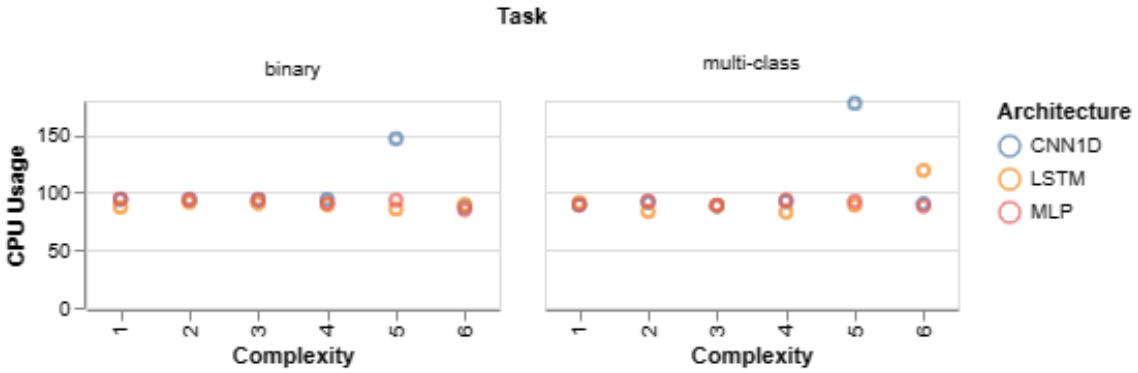


Figure 20: Full-size analysis: CPU usage versus model complexity.

Figure 21: Full-size analysis: power consumption versus model complexity.

inference latency than kin models of lower complexities.

A different perspective on the full-sized models shows how power consumption differs across architectures and complexities (figure 21). MLPs use the least electric current, followed by LSTMs. But the most power-inefficient architecture turns out to be the CNN. Multi-class models tend to have a higher power consumption than their binary counterparts. For energy consumption, we see a change in distribution (figure 22). While MLPs prove to be both power and energy efficient, CNNs use less energy compared to the LSTM due to faster inference. We see that due to lower inference time, some LSTM architectures use less energy even when the number of parameters is higher than other LSTMs.

Our findings point to the fact that different architectures have different optimization needs. We highlight a need for reducing power consumption for CNNs, and reducing inference time and memory require-

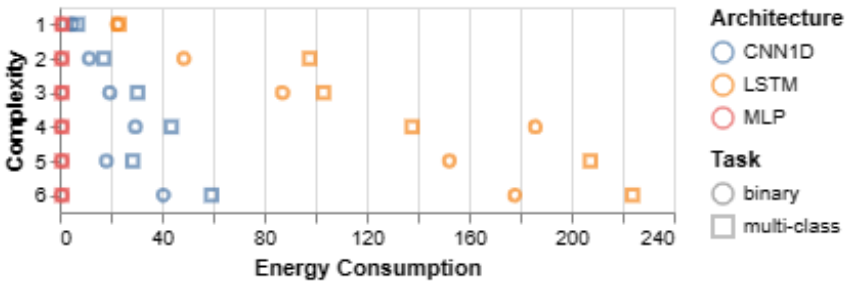


Figure 22: Full-size analysis: energy consumption versus model complexity.

ments for more complex LSTMs. We argue that out of all architectures considered, MLPs are the most resource-efficient, however their accuracy tends to be on the lower side for time-series analysis. Finding a middle-way solution that efficiently addresses all issues above across architecture and complexities without a big compromise in accuracy is thus motivated. Applying such resource-targeted optimizations would evade the necessity of searching for a smaller architecture and trying out multiple optimizations in an exhaustive manner. We move on to the comparative analysis between full-sized and optimized models.

6.2 Comparative Analysis and Evaluation

This subsection showcases the analysis part of the framework which is necessary in order to support our thesis statement. Equipped with empirical knowledge on full-sized architecture challenges and their impact on device requirements, we may begin exploring model optimizations and search for those that fulfill all our success criteria from subsection 2.2. In this thesis we perform this comparative analysis for both tasks (i.e binary and multi-class) separately, however only the binary task will be plotted here, to reduce redundant information. The multi-class experiment points towards the same conclusions. This further proves that even a task with slightly different models used on another randomization of the data set achieves the same results, strengthening our conclusions.

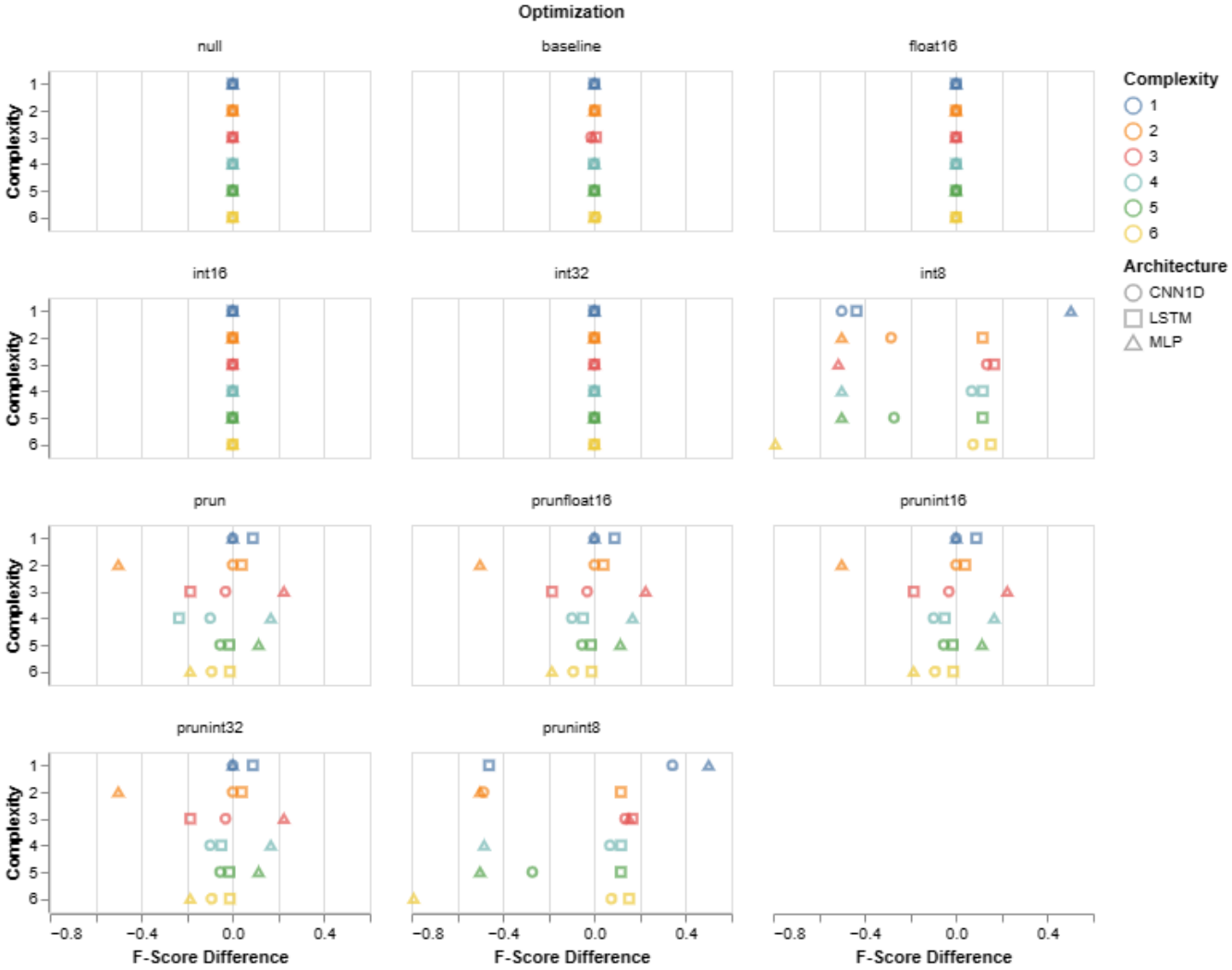


Figure 23: Comparative analysis: F-score difference versus model complexity.

The first success criteria is linked to accurateness. To fulfill it we aim for a optimization set leading to F-score difference values (i.e difference between full-sized and optimized F-scores) close to (given threshold), equal or higher than zero. All F-score differences are found in figure 23. Most purely quantized models stand on the zero line, with the exception of int8 models which distribute values on both sides of the zero line. A less accentuated but similar behaviour is seen for pruning. Pruning employs some retraining of the model, so we actually see that accuracy tends to increase slightly for some joint compression optimizations. Compared to int8, individual pruning and pruned combinations keep values closer to zero, and tend to be more robust as we see a similar shift in values for all architectures. We find that for all pruned models MLP_2 experiences a high drop in F-score. Some LSTM variants are also impacted when pruned and the drop in accuracy is considered substantial (approximately 0.2). Even if int8 optimized binary MLP_1 reaches an increase by approximately 0.5 points, all other binary MLPs are negatively affected. Interestingly, larger LSTMs achieve a better accuracy when quantized by int8, which remains the case even after pruning. One may assume that overparametrized LSTMs are less affected by this optimization, but this pattern is not seen in the multi-class experiment so this reason is not founded.

We argue that quantization methods (excluding int8) are most practical, however if the use case allows a drop in accuracy, individual pruning and/or joint compression (excluding prunint8) could even reach higher accuracies than their full-sized counterparts (CNNs in particular are better performers after applying this optimization). Sometimes int8 is the only possible approach if the hardware itself and supported operations do not allow other precisions. However, our approach has to consider all architectures and a good solution across all model sizes.

Consequently, int8 optimization and pruning strategies are removed from the set of candidate optimizations. Note that we make this elimination based on an assumption that we cannot allow high drops in accuracy (criteria threshold of 0.1), as we would end up with models that differ a lot from their full-sized counterparts in terms of performance.

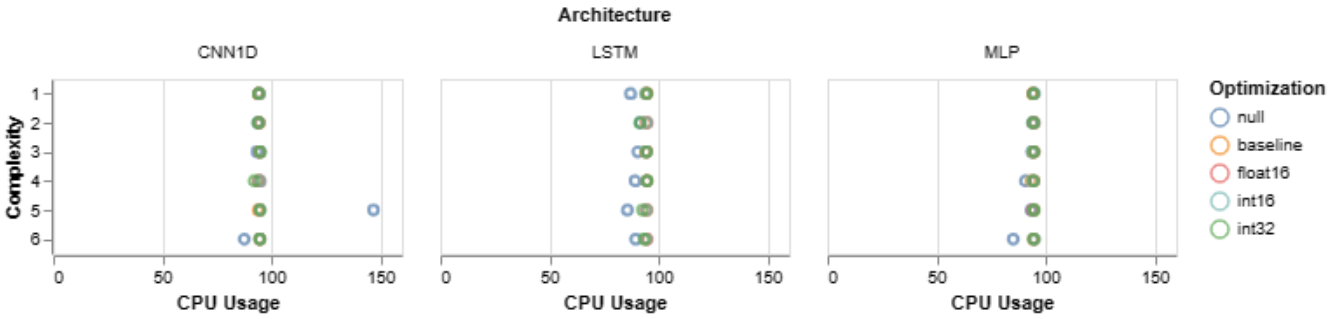


Figure 24: Comparative analysis: CPU usage versus model complexity.

The second criteria is the resource criteria, that is, the optimization should minimize resource utilization so that it is at least below that of the full-size model. In previous sections we have put efforts into finding those resources which are typically targeted in edge environments. We thus list these resources as: computational (CPU Usage), power consumption, memory allocation, inference time and energy consumption. In order to investigate the trade-off between these and accuracy, we will also plot these against the F-score.

In figure 24 we are plotting the CPU utilization values across architectures and across complexities for multiple optimizations, including full-sized models as well (i.e null). We find that all optimizations use less than one CPU core for inference. This means that even the spikes seen previously (e.g CNN_5) can be reduced so that only one core is performing operations. An interesting observation is that full-sized LSTMs use less CPU than all optimized variations, however the gap is very small. We then investigate the relationship between CPU usage and prediction performance across architectures and complexities, for all model types (see figure 25). What we aim for here is finding those optimizations that do not decrease F-score substantially and are equal or lower in CPU utilization when compared to the full-sized values. We see that the baseline, float16, int16, int32 techniques are safe choices in this regard.

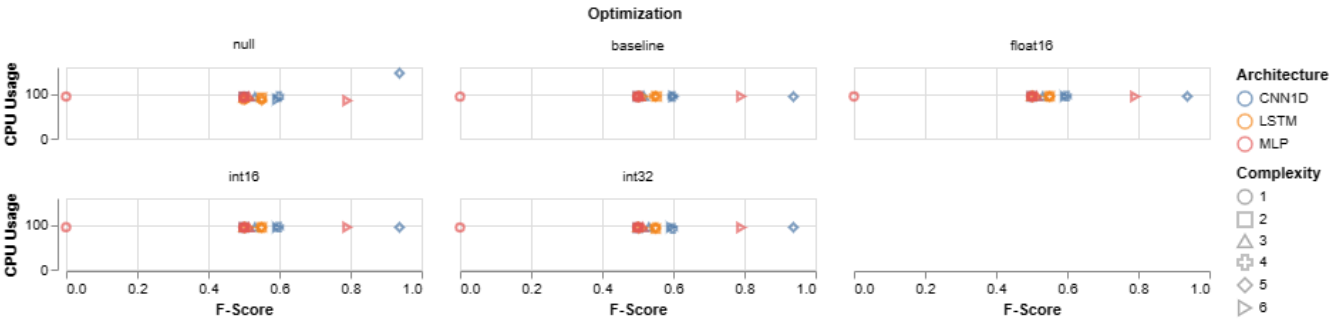


Figure 25: Comparative analysis: CPU usage versus F-score.

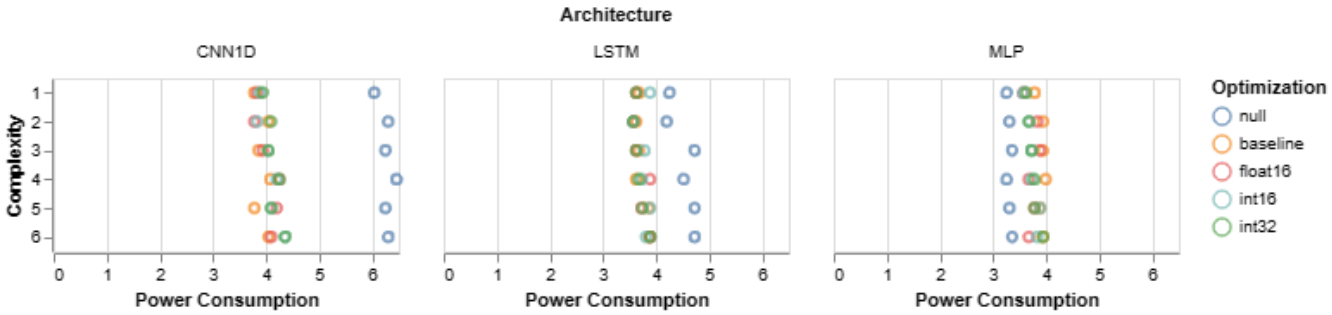


Figure 26: Comparative analysis: power consumption versus model complexity.

Next, we move to power consumption (see figure 26). We observe that power consumption for the full-sized MLP is the lowest across all model sizes, and that optimizations use slightly more power. This could be due to inner working of the interpreter, since it has a different prediction mechanism. However, full-sized LSTMs and especially CNNs register a clear power consumption reduction if optimized, which is true across all complexities. We find that the range of power consumption for all optimized models lays between approximately 3.5W to 4.5W. Since the power consumption difference between MLPs and the optimized models is not big and we have determined that MLPs are the most power-efficient out of all considered architectures, we choose to keep remaining optimizations in the set of potential candidates. In figure 27 we use a similar plot to check F-score against power consumption, where see no significant loss in prediction performance for any of the remaining optimizations.

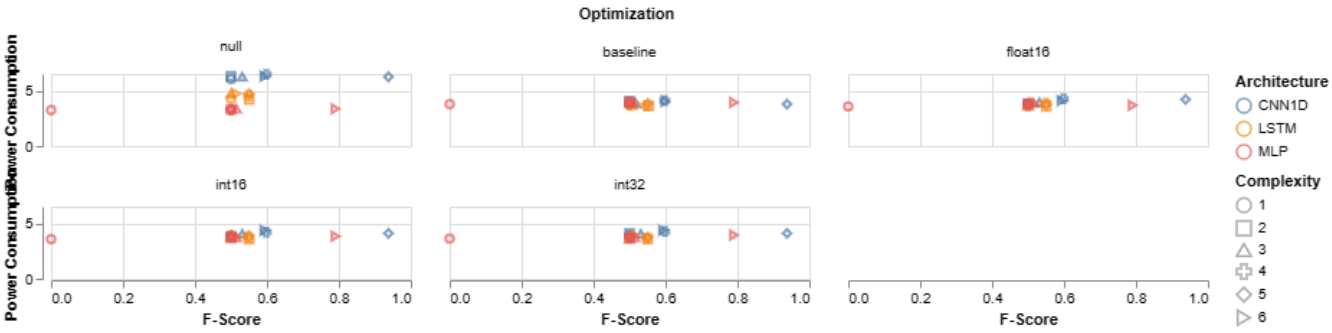


Figure 27: Comparative analysis: power consumption versus F-score.

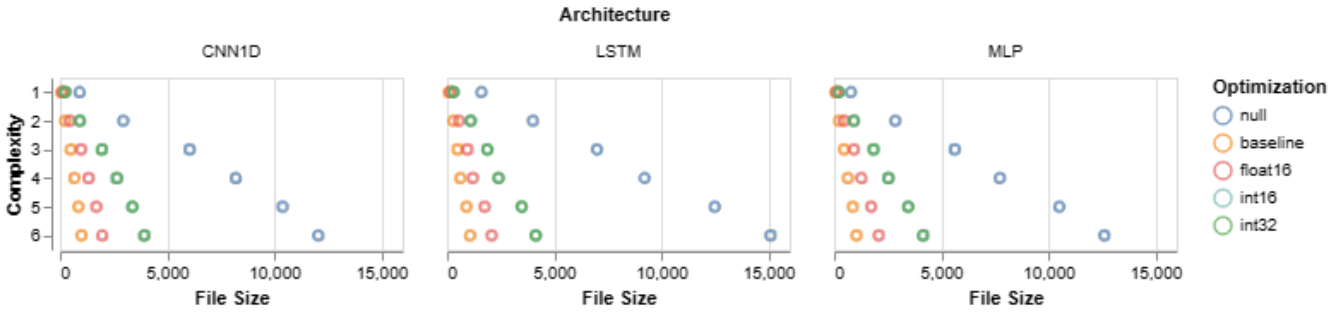


Figure 28: Comparative analysis: file size versus model complexity.

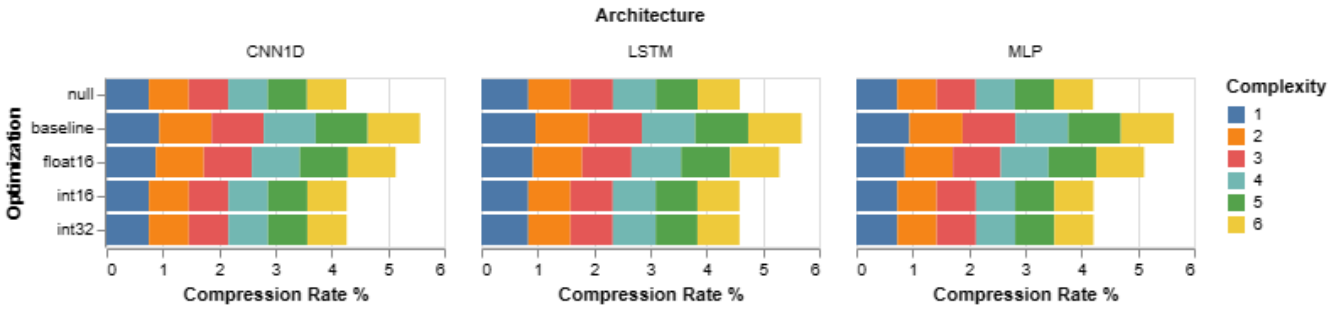


Figure 29: Comparative analysis: compression rate for each model.

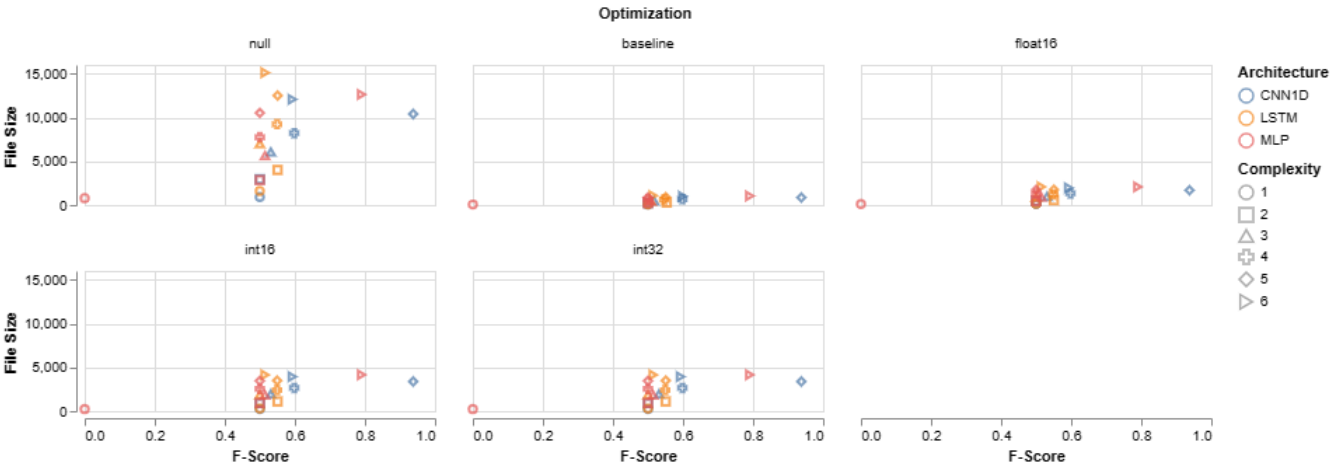


Figure 30: Comparative analysis: file size versus F-score.

Next on the resource list is memory allocation. A model that is too big cannot be loaded onto a device, and so reducing the model file size is imperative for deployment on constricted edge platforms. As can be clearly nuanced in figure 28, across all model architectures and complexities, our remaining optimizations manage to achieve better file sizes. Notice that optimized LSTM files in particular are reduced substantially from their full-sized counterparts, coming close to the optimized MLP and CNN sizes. All of these models, including full-sized ones, can be further compressed. We calculate compression rate (compressed file size divided by the file size of the full-sized model counterpart), and see which optimizations manage to achieve highest compression (see figure 29). We note that the compression power is superior for the baseline and float16 compression techniques. Baseline performs best, since it allows even 8-bit precisions by its dynamic nature. An interesting find is that int16 achieves a worse compression rate than float16. One explanation could be attributed to the compression algorithm and the way it processes 16-bit float values. Note that compressed full-sized models achieve very similar if not equal compression rate to int16 and int32 optimized models, so we prioritize baseline and float16 optimizations as they lead to smaller models. In figure 30 we follow the same steps as before and set up a plot where F-scores are mapped

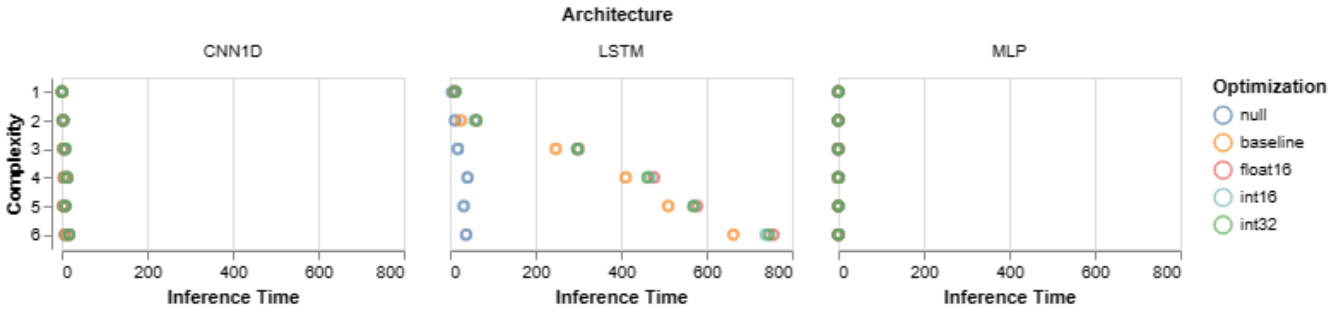


Figure 31: Comparative analysis: inference time versus model complexity.

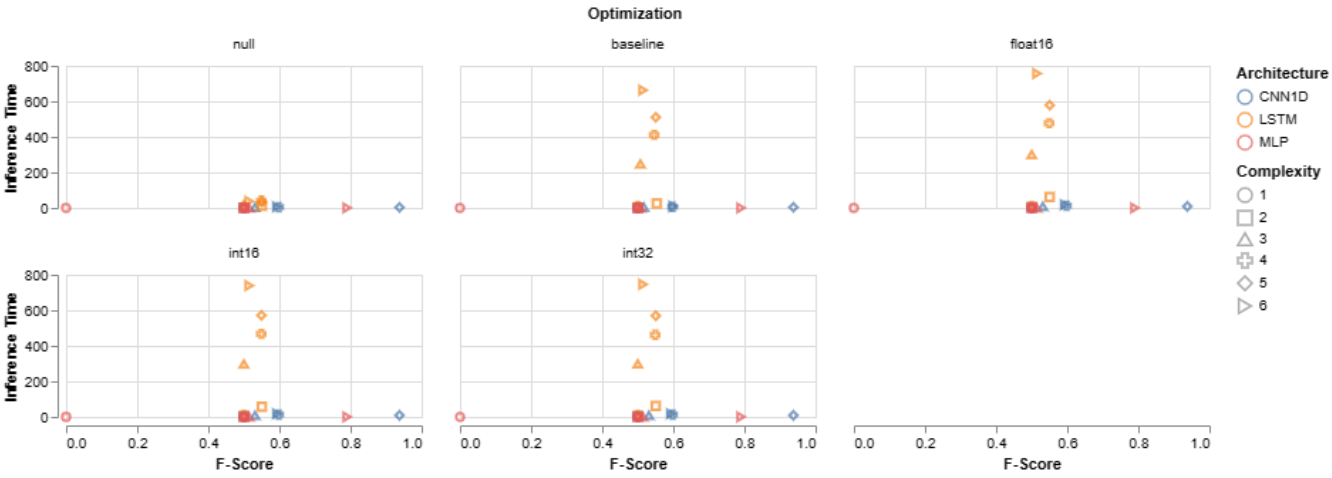


Figure 32: Comparative analysis: inference time versus F-score.

against file size values. Our goal here is to make sure that prediction performance is not lost by optimizing models, and only file size becomes lower. We agree that this is the case for all optimizations.

We continue by plotting inference time (in figure 31). A first remark is that optimized MLPs and CNNs inference time is comparable to that of the full-sized model. But for LSTMs, it is clear that optimization leads to an increase in time needed for inference, especially for more complex models. We believe a reason for this might be the addition of conversion operations between quantized and full-precision values at runtime. Another reason may be the way that Raspberry Pi does its memory accesses for LSTM operations. Some predictions over the infer-

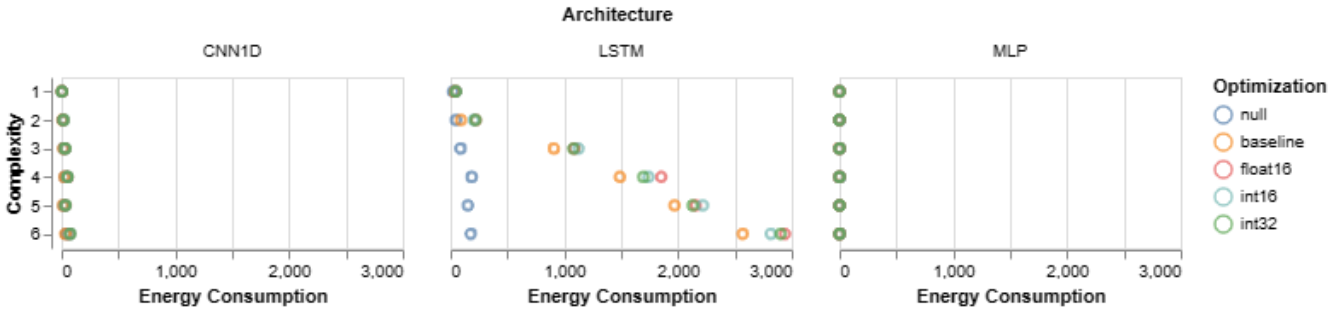


Figure 33: Comparative analysis: energy consumption versus model complexity.

ence set take more than 5 minutes over the full-sized time for LSTMs. No matter which reason, whether these models are practical depends indeed on the use case and whether the system is time-critical, however the inference time is high enough for us to conclude that no optimization has managed to achieve less resource utilization than their full-sized counterparts without much loss of accuracy across all architectures and complexities. In figure 32 we plot the F-score against inference time.

A similar plot is found for the energy consumption, due to the way this metric is calculated (see figure 33). We can also plot the F-score against energy consumption (see figure 34), however for both views we get a similar distribution to what we have seen before for the inference time, drawing the same conclusion.

All in all, we have seen how using our success criteria has served as a proximate factor for filtering out (or retaining) techniques according to whether they fulfill these criteria. The trade-offs between individual resources and predictive performance have also been framed within our analysis. A larger discussion on how these results can be interpreted, challenges and final conclusions drawn from this analysis is necessary.

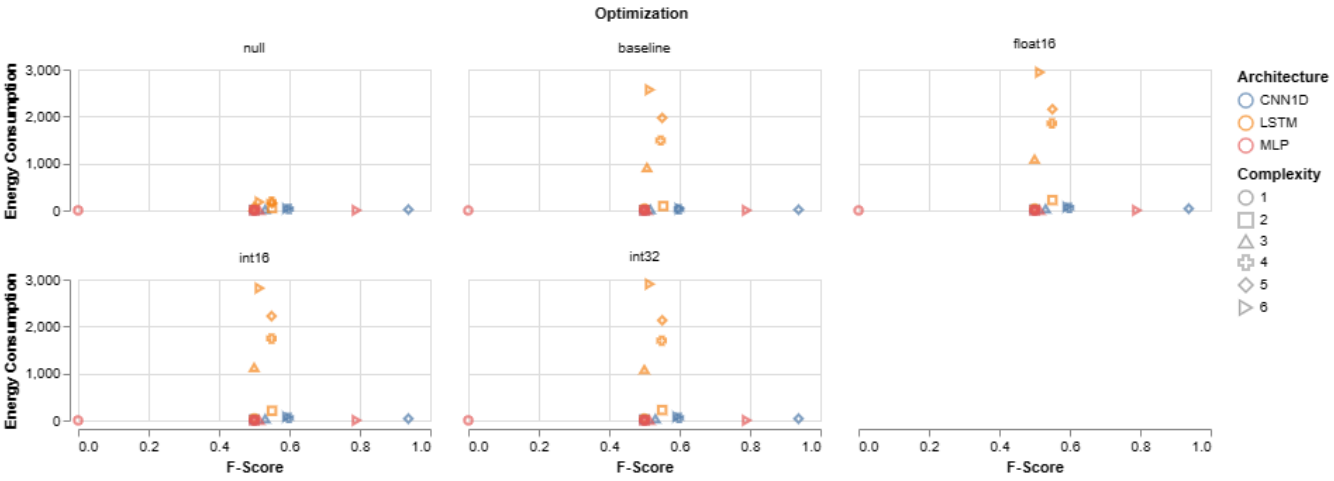


Figure 34: Comparative analysis: energy consumption versus F-score.

7 Conclusion

This last section includes some retrospective insights about the thesis progress so far. We conclude our thesis with a short discussion on the findings from the previous section, communicate encountered limitations to our approach and create a plan for future work.

7.1 Discussion

Every step in the construction of this PoC has been motivated by previous works towards edge inference optimization, investigating status quo and aiming to bridge over encountered research gaps in current literature. Our efforts aim to advance research towards achieving autonomous resource-aware optimization by searching for a practical one-fits-all solution for ANN optimization for edge deployment and inference. By comparing several deep learning model architectures of various structural complexities to their optimized counterparts, we enable a deeper exploration of the trade-offs between prediction performance and resource utilization, which is an important aspect of this thesis. We have set a couple requirements in the form of success criteria which tell us what our framework should be and do. The framework is completed by performing the comparison analysis seen in the previous sec-

tion, however it also includes the steps leading to our design decisions.

In our last section we have commenced with a detailed analysis of full-sized models for both tasks, and how their performance is registered at deployment to the edge device. This analysis is done in order to explore areas in which these architectures challenge the resource requirements of the device. Our proposed solution should address all encountered challenges, and so evidencing what these challenges are is vital in understanding the benefits to our approach. In this analysis we observe that all considered architectures have different demands regarding the available edge device resources. We have found that CNN models are characterized by higher power consumption, while LSTMs are characterized by high memory utilization, inference time and energy consumption. Full-sized MLPs have less resource demands, but their accuracy variation under optimization will prove to be the most unstable. Current literature does not explore solutions over all of these dimensions, but focuses typically on each challenge separately.

The comparative analysis is a core component to our PoC and helps the search for general solutions. We use the success criteria to evaluate whether the candidate optimizations picked by exploring common strategies in current literature fulfill the requirements. Searching for an optimization set which leads to lightweight models using less resources over all considered dimensions proves to be difficult, as optimizations themselves introduce some resource demands. For example, while quantization techniques such as baseline, float16, int16 and int32 provided efficient optimization with regards to accuracy versus CPU usage, power consumption, and memory utilization, they introduced substantial latency and implicitly energy consumption for LSTMs. We are thus talking not only about a accuracy versus resource trade-off, but also a resource versus resource trade-off, under optimization.

Regarding accuracy alone, there are optimizations which have maintained full-size performance, but there are also a few that have superseded it thanks to some retraining (i.e pruning). Indeed, some have also introduced high accuracy drops, but one main reason to this could be that the models haven't had enough retraining to recover accuracy

(e.g int8). From the beginning we have made an assumption that optimized model accuracy must not drop beyond a 0.1 mark. We however agree that this value depends on the application, but also on the initial full-size model accuracy. If one can allow a higher drop, then pruning models could of have been kept in the candidate set.

Another assumption has been made for what we consider to be a practical model in terms of latency. While MLPs and CNNs benefit from our last set of optimizations (using baseline, float16 especially, but also int16 and int32), LSTM latency was much higher for the optimized models than its full-size version. Our initial goal was to have it optimally below that of the full-size model, and so because of this our last optimization set did not fulfill the resource criteria. However, as before, what is regarded a practical in a real setting depends on the application, and whether or not the system is time-critical. If one can allow a higher latency, then these optimizations could of have remained as candidates.

Our PoC is not use case dependent and attempts to provide a general optimization set. While our findings find no such set in the models that we have provided, we strongly believe that this PoC has the potential to achieve this goal. We see in fact that for the CNN and MLP, baseline, float16, int16 and int32 strategies achieve resource-efficient inference across all model complexities. We will thus present some of the limitations of our approach and also pave the way for future work that improves the current framework. This thesis thus contributes not only to enlarging the stock of knowledge around more efficient edge platform inference, but offers a first exploration step in this direction, building a road to autonomous AI.

7.2 Limitations and Future Work

We register a few areas in which our framework could use more improvement.

- We note that it is important to work with stable models. In our experiment we find that the optimized versions of full-sized models with low initial prediction performance tend to be subjected to

higher negative F-score differences than models where this performance is higher. This pattern is observed comparing the binary and multi-class tasks. To avoid this issue, our model generation phase registers the introduction of early stopping and drop-out regularization to avoid overfitting. In future work it would be interesting to approach this from a transfer learning perspective, where one deploys models that already have been trained, with some eventual tuning to fit the use case.

- This thesis limits itself to analysis over neural networks, however exploring inference for traditional models (such as logistic regression, support vector machine, decision trees, naive Bayes) may also offer interesting insights. We considered three architectures due to their popularity. Initial steps in expanding our approach have already been implemented in our code, where we include the model generation code for RNNs and two-dimensional CNNs.

- Our thesis has chosen the two most popular optimization techniques, however adding several variations over these techniques or including other (e.g. knowledge distribution, tensor decomposition) increases the possibility of finding a general set of optimizations. However it also increases the dimension of the study, and so we argue that an exhaustive approach is avoided by prioritizing common techniques. We have provided a throughout taxonomy over all visited optimizations, which is a first step for future work considerations.

- Being a PoC, our deployment environment constitutes of one device. Deployment to multiple devices with different configurations would add a third dimension to the thesis goal: a general set of optimizations across architectures, model complexities and devices. This is indeed an exciting area to explore, to achieve even better generalization. We remind that in this thesis we have enumerated which edge devices are typically targeted for deployment, and so a first step into exploring this area is provided.

- Our approach considers the popular performance indicators, however there are many variations as to how these can be computed. For example, MACs/FLOPs are commonly used for measuring computational load, but we chose to use CPU usage as an alternative measure for how running inference affects computational power on device. Future work should include further exploration of performance metrics to get improved estimations about resource utilization.

Future work should thus be dedicated to completing the above-mentioned tasks, as they are aimed towards a more robust framework.

Appendix A

	Number Participants	*Collection Methods*	*Dataset Size*	*Labeled / unlabeled*	*Problem Types*	*Measurements / Observations*	*Data Type*	*Data Format*
Empatica	- EEG: 14 participants - E4: 11 participants	- 8-channel OpenBCI headset - Empatica E4 wristband	86.06 MB	Labeled by FAS scores (fatigue state score)	Fatigue detection	- Electroencephalographic (EEG) signals - Photoplethysmography (PPG) - HR: Heart rate - IBI: Inter-beat interval - TEMP: Skin temperature - EDA: Electrodermal Activity - BVP: Blood Volume Pulse - ACC: Accelerometer data - SRates: Sampling Rates of the ACC, BVP, EDA, HR, IBI and TEMP	Time-series	.zip (included .mat) and matlab file
Toadstool	10 participants (5 female, 5 male)	- Empatica E4 wristband - Camera:a 1.3-MP webcam of a Samsung Series 9 Notebook NP900X4C	7.5 GB	Unlabeled	- Facial expression studies - Sentiment analysis - Game studies - Reinforcement learning - Emotionally-aware ML	- Controller input - Video at 30 frames per second with a resolution of 640 × 480. - HR: Heart rate - IBI: Inter-beat interval - TEMP: Skin temperature - EDA: Electrodermal Activity - BVP: Blood Volume Pulse - ACC: Accelerometer data	- Video files - Time-series	.avi, JSON, .csv, .text
WESAD	15 participants (12 male, 3 female)	- Chest-worn device (RespiBAN) - Empatica E4 wristband.	Instances: 63000000 2.1 GB	Labeled by self-reports on emotional states (PANAS, STAI, SAM, SSSQ approaches: fill-in questions)	- Stress and affect detection - Classification - Regression - Affective computing	- Electrocardiogram (ECG) - Electrodermal activity (EDA) - Electromyogram (EMG) - Respiration - Body temperature - Three-axis acceleration - BVP: Blood Volume Pulse - EDA: Electrodermal Activity - TEMP: Skin temperature - ACC: Accelerometer data	- Multivariate - Time-series	
GNOMON	15 participants (all female)	- Fitbit Charge 5		Labeled by self-reports (fatigue score)	Fatigue and stress detection	- Stress Score - Physical Activity - Menstrual Health Metrics - Fitness fatigue component - HR: Heart rate - Sleep Score - Others (i.e. blood glucose)	Time-series	.csv, JSON

Figure 35: Candidate data sets

References

Aceto, G., Persico, V., & Pescapé, A. (2020). Industry 4.0 and health: Internet of things, big data, and cloud computing for healthcare 4.0. *Journal of Industrial Information Integration*, *18*, 100129.

Aha, D. W. (2013). *Lazy learning*. Springer Science & Business Media.

Ay, E., Devanne, M., Weber, J., & Forestier, G. (2022). A study of knowledge distillation in fully convolutional network for time series classification. *2022 International Joint Conference on Neural Networks (IJCNN)*, 1–8.

Bahrebar, P., Denis, L., Bonnaerens, M., Coddens, K., Dambre, J., Favoreel, W., Khvastunov, I., Munteanu, A., Nguyen-Duc, H., Schulte, S., et al. (2021). Creative: Reconfigurable embedded artificial intelligence. *Proceedings of the 18th ACM International Conference on Computing Frontiers*, 194–199.

Bai, Y., Guan, Y., & Ng, W.-F. (2020). Fatigue assessment using ecg and actigraphy sensors. *Proceedings of the 2020 International Symposium on Wearable Computers*, 12–16.

Bangaru, S. S., Wang, C., & Aghazadeh, F. (2022). Automated and continuous fatigue monitoring in construction workers using forearm emg and imu wearable sensors and recurrent neural network. *Sensors*, *22*(24), 9729.

Bannigan, P., Aldeghi, M., Bao, Z., Häse, F., Aspuru-Guzik, A., & Allen, C. (2021). Machine learning directed drug formulation development. *Advanced Drug Delivery Reviews*, *175*, 113806.

Bian, S., Wang, X., Polonelli, T., & Magno, M. (2022). Exploring automatic gym workouts recognition locally on wearable resource-constrained devices. *2022 IEEE 13th International Green and Sustainable Computing Conference (IGSC)*, 1–6.

Bína, V., Bartošová, J., & Přibyl, V. (2022). Anomaly detection in time series for smart agriculture. *International Journal of Management, Knowledge and Learning*, *11*.

Bohra, S., Naik, V., & Yeligar, V. (2021). Towards putting deep learning on the wrist for accurate human activity recognition. *2021 IEEE International Conference on Pervasive Computing and Communications Workshops and other Affiliated Events (PerCom Workshops)*, 605–610.

Burrello, A., Pagliari, D. J., Rapa, P. M., Semilia, M., Risso, M., Polonelli, T., Poncino, M., Benini, L., & Benatti, S. (2022). Embedding temporal convolutional networks for energy-efficient ppg-based heart rate monitoring. *ACM Transactions on Computing for Healthcare (HEALTH)*, *3*(2), 1–25.

Burrello, A., Pagliari, D. J., Risso, M., Benatti, S., Macii, E., Benini, L., & Poncino, M. (2021). Q-ppg: Energy-efficient ppg-based heart rate monitoring on wearable devices. *IEEE Transactions on Biomedical Circuits and Systems*, *15*(6), 1196–1209.

Buyya, R., Broberg, J., & Goscinski, A. M. (2010). *Cloud computing: Principles and paradigms*. John Wiley & Sons.

Cerina, L., Santambrogio, M. D., Franco, G., Gallicchio, C., & Micheli, A. (2020). Efficient embedded machine learning applications using echo state networks. *2020 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, 1299–1302.

Chandna, M., Bhatia, P., Singh, S.-P., & Suneja, S. (2023). Tiny face presence detector using hybrid binary neural network. *2023 4th International Conference on Innovative Trends in Information Technology (ICITIIT)*, 1–5.

Chatterjee, D., Dutta, S., Shaikh, R., & Saha, S. K. (2022). A lightweight deep neural network for detection of mental states from physiological signals. *Innovations in Systems and Software Engineering*, 1–8.

Chauhan, J., Rajasegaran, J., Seneviratne, S., Misra, A., Seneviratne, A., & Lee, Y. (2018). Performance characterization of deep learning models for breathing-based authentication on resource-constrained devices. *Proceedings of the ACM on Interactive, Mobile, Wearable and Ubiquitous Technologies*, *2*(4), 1–24.

Chen, J., & Ran, X. (2019). Deep learning with edge computing: A review. *Proceedings of the IEEE*, *107*(8), 1655–1674.

Cheng, X., Zhang, L., Tang, Y., Liu, Y., Wu, H., & He, J. (2022). Real-time human activity recognition using conditionally parametrized convolutions on mobile and wearable devices. *IEEE Sensors Journal*, *22*(6), 5889–5901.

Cheng, Y., Huang, G., Zhen, P., Liu, B., Chen, H.-B., Wong, N., & Yu, H. (2020). An anomaly comprehension neural network for surveillance videos on terminal devices. *2020 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, 1396–1401.

Cheng, Y., Li, G., Wong, N., Chen, H.-B., & Yu, H. (2020). Deepeye: A deeply tensor-compressed neural network for video comprehension on terminal devices. *ACM Transactions on Embedded Computing Systems (TECS)*, *19*(3), 1–25.

Cho, K., Van Merriënboer, B., Gulcehre, C., Bahdanau, D., Bougares, F., Schwenk, H., & Bengio, Y. (2014). Learning phrase representations using rnn encoder-decoder for statistical machine translation. *arXiv preprint arXiv:1406.1078*.

Christ, G. (2016). Burnt Out: Stress on the Job [Infographic] [[Online; accessed 7-June-2022]].

Cipra, T. (2020). *Time series in economics and finance*. Springer Nature.

Coelho, Y. L., dos Santos, F. d. A. S., Frizera-Neto, A., & Bastos-Filho, T. F. (2021). A lightweight framework for human activity recognition on wearable devices. *IEEE Sensors Journal*, *21*(21), 24471–24481.

Coelho Jr, C. N., Kuusela, A., Li, S., Zhuang, H., Ngadiuba, J., Aarrestad, T. K., Loncar, V., Pierini, M., Pol, A. A., & Summers, S. (2021). Automatic heterogeneous quantization of deep neural networks for low-latency inference on the edge for particle detectors. *Nature Machine Intelligence*, *3*(8), 675–686.

Daghero, F., Burrello, A., Xie, C., Castellano, M., Gandolfi, L., Calimera, A., Macii, E., Poncino, M., & Pagliari, D. J. (2022). Human activity recognition on microcontrollers with quantized and adaptive deep neural networks. *ACM Transactions on Embedded Computing Systems (TECS)*, *21*(4), 1–28.

Daghero, F., Xie, C., Pagliari, D. J., Burrello, A., Castellano, M., Gandolfi, L., Calimera, A., Macii, E., & Poncino, M. (2021). Ultra-compact binary neural networks for human activity recognition on risc-v processors. *Proceedings of the 18th ACM International Conference on Computing Frontiers*, 3–11.

De Vita, F., Nocera, G., Bruneo, D., Tomaselli, V., Giacalone, D., & Das, S. K. (2020). Quantitative analysis of deep leaf: A plant disease detector on the smart edge. *2020 IEEE International Conference on Smart Computing (SMARTCOMP)*, 49–56.

Dey, E., & Roy, N. (2020). Omad: On-device mental anomaly detection for substance and non-substance users. *2020 IEEE 20th International Conference on Bioinformatics and Bioengineering (BIBE)*, 466–471.

Dhar, S., Guo, J., Liu, J., Tripathi, S., Kurup, U., & Shah, M. (2021). A survey of on-device machine learning: An algorithms and learning theory perspective. *ACM Transactions on Internet of Things*, *2*(3), 1–49.

Escobar-Linero, E., Domınguez-Morales, M., & Sevillano, J. L. (2022). Worker's physical fatigue classification using neural networks. *Expert Systems with Applications*, *198*, 116784.

Faraone, A., & Delgado-Gonzalo, R. (2020). Convolutional-recurrent neural networks on low-power wearable platforms for cardiac arrhythmia detection. *2020 2nd IEEE International Conference on Artificial Intelligence Circuits and Systems (AICAS)*, 153–157.

Faraone, A., Sigurthorsdottir, H., & Delgado-Gonzalo, R. (2021). Atrial fibrillation detection on low-power wearables using knowledge distillation. *2021 43rd Annual International Conference of the IEEE Engineering in Medicine & Biology Society (EMBC)*, 6795–6799.

Fard, M. J., Ameri, S., Darin Ellis, R., Chinnam, R. B., Pandya, A. K., & Klein, M. D. (2018). Automated robot-assisted surgical skill evaluation: Predictive analytics approach. *The International Journal of Medical Robotics and Computer Assisted Surgery*, *14*(1), e1850.

Garcia-Ceja, E., Osmani, V., & Mayora, O. (2015). Automatic stress detection in working environments from smartphones' accelerometer data: A first step. *IEEE journal of biomedical and health informatics*, *20*(4), 1053–1060.

Ghasemzadeha, M., Mohammad-Karimi, N., & Ansari-Samani, H. (2020). Machine learning algorithms for time series in financial markets. *Advances in Mathematical Finance and Applications*, *5*(4), 479–490.

Ghosh, S., Kim, S., Ijaz, M. F., Singh, P. K., & Mahmud, M. (2022). Classification of mental stress from wearable physiological sensors using image-encoding-based deep neural network. *Biosensors*, *12*(12), 1153.

Goodfellow, I., Bengio, Y., & Courville, A. (2016). *Deep learning* [http://www.deeplearningbook.org]. MIT Press.

Google. (2022). *Classification: Thresholding* [Accessed: 2023-01-25]. https://developers.google.com/machine-learning/crash-course/classification/thresholding

Graves, A. (2012). *Supervised sequence labelling*. Springer.

Gupta, S., Imani, M., Zhao, H., Wu, F., Zhao, J., & Rosing, T. Š. (2020). Implementing binary neural networks in memory with approximate accumulation. *Proceedings of the ACM/IEEE International Symposium on Low Power Electronics and Design*, 247–252.

Hassan, A., & Mahmood, A. (2017). Deep learning approach for sentiment analysis of short texts. *2017 3rd international conference on control, automation and robotics (ICCAR)*, 705–710.

Hillier, D., Fewell, F., Cann, W., & Shephard, V. (2005). Wellness at work: Enhancing the quality of our working lives. *International Review of Psychiatry*, *17*(5), 419–431.

Hiremath, S., Yang, G., & Mankodiya, K. (2014). Wearable internet of things: Concept, architectural components and promises for person-centered healthcare. *2014 4th International Conference on Wireless Mobile Communication and Healthcare-Transforming Healthcare Through Innovations in Mobile and Wireless Technologies (MOBIHEALTH)*, 304–307.

Hochreiter, S., & Schmidhuber, J. (1997). Long short-term memory. *Neural computation*, *9*(8), 1735–1780.

Hornik, K., Stinchcombe, M., & White, H. (1989). Multilayer feedforward networks are universal approximators. *Neural networks*, *2*(5), 359–366.

Huang, A., & Wu, R. (2016). Deep learning for music. *arXiv preprint arXiv:1606.04930*.

Hussain, B., Afzal, M. K., Ahmad, S., & Mostafa, A. M. (2021). Intelligent traffic flow prediction using optimized gru model. *IEEE Access*, *9*, 100736–100746.

Huynh, L., Nguyen, T., Nguyen, T., Pirttikangas, S., & Siirtola, P. (2021). Stressnas: Affect state and stress detection using neural architecture search. *Adjunct Proceedings of the 2021 ACM International Joint Conference on Pervasive and Ubiquitous Computing and Proceedings of the 2021 ACM International Symposium on Wearable Computers*, 121–125.

IEEE. (2022). Top Technology Trends for 2023 [[Online; accessed 28-March-2023]].

Intel. (n.d.). Deep Learning Inference at Scale with AI Readiness [[Online; accessed 28-March-2023]].

International Labour Office. (2019). Safety and health at the heart of the future of work: Building on 100 years of experience, 1–68.

Iqbal, S., Siddiqui, G. F., Rehman, A., Hussain, L., Saba, T., Tariq, U., & Abbasi, A. A. (2021). Prostate cancer detection using deep learning and traditional techniques. *IEEE Access*, *9*, 27085–27100.

Jaeger, H. (2001). The "echo state" approach to analysing and training recurrent neural networks-with an erratum note. *Bonn, Germany: German National Research Center for Information Technology GMD Technical Report*, *148*(34), 13.

Kagermann, H., Lukas, W.-D., & Wahlster, W. (2011). Industrie 4.0: Mit dem internet der dinge auf dem weg zur 4. industriellen revolution. *VDI nachrichten*, *13*(1), 2–3.

Kalgaonkar, P., & El-Sharkawy, M. (2021). Condensenext: An ultra-efficient deep neural network for embedded systems. *2021 IEEE 11th Annual Computing and Communication Workshop and Conference (CCWC)*, 0524–0528.

Kansara, D., & Sawant, V. (2020). Comparison of traditional machine learning and deep learning approaches for sentiment analysis. *Advanced computing technologies and applications* (pp. 365–377). Springer.

Kaplan, A., & Haenlein, M. (2019). Siri, siri, in my hand: Who's the fairest in the land? on the interpretations, illustrations, and implications of artificial intelligence. *Business Horizons*, *62*(1), 15–25.

Karthikeyan, N. et al. (2019). Mobile artificial intelligence projects.

Kofod-Petersen, A. (2012). How to do a structured literature review in computer science. *Ver. 0.1. October*, *1*.

Kumar, A., Seshadri, V., & Sharma, R. (2020). Shiftry: Rnn inference in 2kb of ram. *Proceedings of the ACM on Programming Languages*, *4*(OOPSLA), 1–30.

Lalapura, V. S., Amudha, J., & Satheesh, H. S. (2021). Recurrent neural networks for edge intelligence: A survey. *ACM Computing Surveys (CSUR)*, *54*(4), 1–38.

Lee, C., & Lim, C. (2021). From technological development to social advance: A review of industry 4.0 through machine learning. *Technological Forecasting and Social Change*, *167*, 120653.

Leppink, N. (2015). Socio-economic costs of work-related injuries and illnesses: Building synergies between occupational safety and health and productivity. *National Institute for Insurance against Accidents at Work, Bologna, Italy*.

Li, S., Man, C., Shen, A., Guan, Z., Mao, W., Luo, S., Zhang, R., & Yu, H. (2022). A fall detection network by 2d/3d spatio-temporal joint models with tensor compression on edge. *ACM Transactions on Embedded Computing Systems*, *21*(6), 1–19.

Li, Y., & Parker, L. E. (2014). Nearest neighbor imputation using spatial–temporal correlations in wireless sensor networks. *Information Fusion*, *15*, 64–79.

Little, B., Alshabrawy, O., Stow, D., Ferrier, I. N., McNaney, R., Jackson, D. G., Ladha, K., Ladha, C., Ploetz, T., Bacardit, J., et al. (2021). Deep learning-based automated speech detection as a marker of social functioning in late-life depression. *Psychological medicine*, *51*(9), 1441–1450.

Liu, D., Kong, H., Luo, X., Liu, W., & Subramaniam, R. (2022). Bringing ai to edge: From deep learning's perspective. *Neurocomputing*, *485*, 297–320.

Liu, G., Zhong, K., Li, H., Chen, T., & Wang, Y. (2022). A state of art review on time series forecasting with machine learning for environmental parameters in agricultural greenhouses. *Information Processing in Agriculture*.

Liu, H. (2013). Big data drives cloud adoption in enterprise. *IEEE internet computing*, *17*(4), 68–71.

Liu, S., Yao, S., Li, J., Liu, D., Wang, T., Shao, H., & Abdelzaher, T. (2020). Giobalfusion: A global attentional deep learning framework for multisensor information fusion. *Proceedings of the ACM on Interactive, Mobile, Wearable and Ubiquitous Technologies*, *4*(1), 1–27.

Liu, S., Guo, B., Ma, K., Yu, Z., & Du, J. (2021). Adaspring: Context-adaptive and runtime-evolutionary deep model compression for mobile applications. *Proceedings of the ACM on Interactive, Mobile, Wearable and Ubiquitous Technologies*, *5*(1), 1–22.

Lu, B., Yang, J., Chen, L. Y., & Ren, S. (2019). Automating deep neural network model selection for edge inference. *2019 IEEE First International Conference on Cognitive Machine Intelligence (CogMI)*, 184–193.

Marsland, S. (2011). *Machine learning: An algorithmic perspective*. Chapman; Hall/CRC.

Mazzei, D., & Ramjattan, R. (2022). Machine learning for industry 4.0: A systematic review using deep learning-based topic modelling. *Sensors*, *22*(22), 8641.

Meyer, M., Farei-Campagna, T., Pasztor, A., Forno, R. D., Gsell, T., Faillettaz, J., Vieli, A., Weber, S., Beutel, J., & Thiele, L. (2019). Event-triggered natural hazard monitoring with convolutional neural networks on the edge. *Proceedings of the 18th International Conference on Information Processing in Sensor Networks*, 73–84.

Mirsalari, S. A., Nazari, N., Ansarmohammadi, S. A., Sinaei, S., Salehi, M. E., & Daneshtalab, M. (2021). Elc-ecg: Efficient lstm cell for ecg classification based on quantized architecture. *2021 IEEE International Symposium on Circuits and Systems (ISCAS)*, 1–5.

Mishra, R., Gupta, H. P., & Dutta, T. (2020). A road health monitoring system using sensors in optimal deep neural network. *IEEE Sensors Journal*, *21*(14), 15527–15534.

Ni, J., Sarbajna, R., Liu, Y., Ngu, A. H., & Yan, Y. (2022). Cross-modal knowledge distillation for vision-to-sensor action recognition. *ICASSP 2022-2022 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, 4448–4452.

Ni, K., Ramanathan, N., Chehade, M. N. H., Balzano, L., Nair, S., Zahedi, S., Kohler, E., Pottie, G., Hansen, M., & Srivastava, M. (2009). Sensor network data fault types. *ACM Transactions on Sensor Networks (TOSN)*, *5*(3), 1–29.

Nielsen, M. A. (2015). *Neural networks and deep learning*. Determination Press.

Nizam, H., Zafar, S., Lv, Z., Wang, F., & Hu, X. (2022). Real-time deep anomaly detection framework for multivariate time-series data in industrial iot. *IEEE Sensors Journal*, *22*(23), 22836–22849.

Pal, A., & Prakash, P. (2017). *Practical time series analysis: Master time series data processing, visualization, and modeling using python*. Packt Publishing Ltd.

Panch, T., Mattie, H., & Celi, L. A. (2019). The "inconvenient truth" about ai in healthcare. *NPJ digital medicine*, *2*(1), 1–3.

Parikh, S., Dave, D., Patel, R., & Doshi, N. (2019). Security and privacy issues in cloud, fog and edge computing. *Procedia Computer Science*, *160*, 734–739.

Patel, V., Chesmore, A., Legner, C. M., & Pandey, S. (2022). Trends in workplace wearable technologies and connected-worker solutions for next-generation occupational safety, health, and productivity. *Advanced Intelligent Systems*, *4*(1), 2100099.

Peluso, V., Cipolletta, A., Calimera, A., Poggi, M., Tosi, F., Aleotti, F., & Mattoccia, S. (2021). Monocular depth perception on microcontrollers for edge applications. *IEEE Transactions on Circuits and Systems for Video Technology*, *32*(3), 1524–1536.

Plarre, K., Raij, A., Hossain, S. M., Ali, A. A., Nakajima, M., Al'Absi, M., Ertin, E., Kamarck, T., Kumar, S., Scott, M., et al. (2011). Continuous inference of psychological stress from sensory measurements collected in the natural environment. *Proceedings of the 10th ACM/IEEE international conference on information processing in sensor networks*, 97–108.

Pouyanfar, S., Sadiq, S., Yan, Y., Tian, H., Tao, Y., Reyes, M. P., Shyu, M.-L., Chen, S.-C., & Iyengar, S. S. (2018). A survey on deep learning: Algorithms, techniques, and applications. *ACM Computing Surveys (CSUR)*, *51*(5), 1–36.

Profentzas, C., Almgren, M., & Landsiedel, O. (2021). Performance of deep neural networks on low-power iot devices. *Proceedings of the workshop on benchmarking cyber-physical systems and Internet of things*, 32–37.

Putri, A. R., Anyanwu, G. O., Maharani, M. P., Lee, J. M., & Kim, D.-S. (2021). Compressed neural network for thermal array-based fall detection system on embedded ai. *2021 International Conference on Information and Communication Technology Convergence (ICTC)*, 1754–1757.

Qian, C., Einhaus, L., & Schiele, G. (2022). Elasticai-creator: Optimizing neural networks for time-series-analysis for on-device machine learning in iot systems. *Proceedings of the 20th ACM Conference on Embedded Networked Sensor Systems*, 941–946.

Ragav, A., Krishna, N. H., Narayanan, N., Thelly, K., & Vijayaraghavan, V. (2019). Scalable deep learning for stress and affect detection on resource-constrained devices. *2019 18th IEEE International Conference On Machine Learning And Applications (ICMLA)*, 1585–1592.

Rai, R., Tiwari, M. K., Ivanov, D., & Dolgui, A. (2021). Machine learning in manufacturing and industry 4.0 applications.

Ribeiro, H. D. M., Arnold, A., Howard, J. P., Shun-Shin, M. J., Zhang, Y., Francis, D. P., Lim, P. B., Whinnett, Z., & Zolgharni, M. (2022). Ecg-based real-time arrhythmia monitoring using quantized deep neural networks: A feasibility study. *Computers in Biology and Medicine*, *143*, 105249.

Risso, M., Burrello, A., Conti, F., Lamberti, L., Chen, Y., Benini, L., Macii, E., Poncino, M., & Pagliari, D. J. (2022). Lightweight neural architecture search for temporal convolutional networks at the edge. *IEEE Transactions on Computers*.

Sahu, R., Dash, S. R., Cacha, L. A., Poznanski, R. R., & Parida, S. (2020). Epileptic seizure detection: A comparative study between deep and traditional machine learning techniques. *Journal of integrative neuroscience*, *19*(1), 1–9.

Sajjad, M., Irfan, M., Muhammad, K., Del Ser, J., Sanchez-Medina, J., Andreev, S., Ding, W., & Lee, J. W. (2020). An efficient and scalable simulation model for autonomous vehicles with economical hardware. *IEEE transactions on intelligent transportation systems*, *22*(3), 1718–1732.

Sakib, S., Fouda, M. M., Fadlullah, Z. M., & Nasser, N. (2020). Migrating intelligence from cloud to ultra-edge smart iot sensor based on deep learning: An arrhythmia monitoring use-case. *2020 International Wireless Communications and Mobile Computing (IWCMC)*, 595–600.

Satyanarayanan, M., Bahl, P., Caceres, R., & Davies, N. (2009). The case for vm-based cloudlets in mobile computing. *IEEE pervasive Computing*, *8*(4), 14–23.

Schmidt, P., Reiss, A., Duerichen, R., Marberger, C., & Van Laerhoven, K. (2018). Introducing wesad, a multimodal dataset for wearable stress and affect detection. *Proceedings of the 20th ACM international conference on multimodal interaction*, 400–408.

Schneider, T., Wang, X., Hersche, M., Cavigelli, L., & Benini, L. (2020). Q-eegnet: An energy-efficient 8-bit quantized parallel eegnet implementation for edge motor-imagery brain-machine interfaces. *2020 IEEE International Conference on Smart Computing (SMARTCOMP)*, 284–289.

Setiowati, S., Franita, E. L., Ardiyanto, I., et al. (2017). A review of optimization method in face recognition: Comparison deep learning and non-deep learning methods. *2017 9th International Conference on Information Technology and Electrical Engineering (ICITEE)*, 1–6.

Shafique, M., Theocharides, T., Bouganis, C.-S., Hanif, M. A., Khalid, F., Hafız, R., & Rehman, S. (2018). An overview of next-generation architectures for machine learning: Roadmap, opportunities and challenges in the iot era. *2018 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, 827–832.

Shaw, J., Rudzicz, F., Jamieson, T., Goldfarb, A., et al. (2019). Artificial intelligence and the implementation challenge. *Journal of medical Internet research*, *21*(7), e13659.

Shruthi, U., Nagaveni, V., & Raghavendra, B. (2019). A review on machine learning classification techniques for plant disease detection. *2019 5th International conference on advanced computing & communication systems (ICACCS)*, 281–284.

Shuvo, M. M. H., Islam, S. K., Cheng, J., & Morshed, B. I. (2022). Efficient acceleration of deep learning inference on resource-constrained edge devices: A review. *Proceedings of the IEEE*.

Siccoli, A., Marlies, P., Schröder, M. L., & Staartjes, V. E. (2019). Machine learning–based preoperative predictive analytics for lumbar spinal stenosis. *Neurosurgical Focus*, *46*(5), E5.

Sun, H., Wang, A., Pu, N., Li, Z., Huang, J., Liu, H., & Qi, Z. (2021). Arrhythmia classifier using convolutional neural network with adaptive loss-aware multi-bit networks quantization. *2021 2nd International Conference on Artificial Intelligence and Computer Engineering (ICAICE)*, 461–467.

Svertoka, E., Saafi, S., Rusu-Casandra, A., Burget, R., Marghescu, I., Hosek, J., & Ometov, A. (2021). Wearables for industrial work safety: A survey. *Sensors*, *21*(11), 3844.

Takagaki, K., Arai, H., & Takayama, K. (2010). Creation of a tablet database containing several active ingredients and prediction of their pharmaceutical characteristics based on ensemble artificial neural networks. *Journal of pharmaceutical sciences*, *99*(10), 4201–4214.

Tang, C. S., & Veelenturf, L. P. (2019). The strategic role of logistics in the industry 4.0 era. *Transportation Research Part E: Logistics and Transportation Review*, *129*, 1–11.

Tang, S. Y., Hoang, N. S., Chui, C. K., Lim, J. H., & Chua, M. C. (2019). Development of wearable gait assistive device using recurrent neural network. *2019 IEEE/SICE International Symposium on System Integration (SII)*, 626–631.

Teh, H. Y., Kempa-Liehr, A. W., & Wang, K. I.-K. (2020). Sensor data quality: A systematic review. *Journal of Big Data*, *7*(1), 1–49.

Tensorflow. (2022). Pruning in keras example [[Online; accessed 30-April-2023]]. https://www.tensorflow.org/model_optimization/guide/pruning/pruning_with_keras

Tesfai, H., Saleh, H., Al-Qutayri, M., Mohammad, M. B., Tekeste, T., Khandoker, A., & Mohammad, B. (2022). Lightweight shufflenet based cnn for arrhythmia classification. *IEEE Access*, *10*, 111842–111854.

Thompson, N. C., Greenewald, K., Lee, K., & Manso, G. F. (2020). The computational limits of deep learning. *arXiv preprint arXiv:2007.05558*.

Thomson, W., Bhowmik, N., & Breckon, T. P. (2020). Efficient and compact convolutional neural network architectures for non-temporal

real-time fire detection. *2020 19th IEEE International Conference on Machine Learning and Applications (ICMLA)*, 136–141.

Uddin, M. T., & Canavan, S. (2019). Synthesizing physiological and motion data for stress and meditation detection. *2019 8th International Conference on Affective Computing and Intelligent Interaction Workshops and Demos (ACIIW)*, 244–247.

Ukil, A., Sahu, I., Majumdar, A., Racha, S. C., Kulkarni, G., Choudhury, A. D., Khandelwal, S., Ghose, A., & Pal, A. (2021). Resource constrained cvd classification using single lead ecg on wearable and implantable devices. *2021 43rd Annual International Conference of the IEEE Engineering in Medicine & Biology Society (EMBC)*, 886–889.

Van Nguyen, T., An, J., & Min, K.-S. (2021). Comparative study on quantization-aware training of memristor crossbars for reducing inference power of neural networks at the edge. *2021 International Joint Conference on Neural Networks (IJCNN)*, 1–6.

Wang, X., Magno, M., Cavigelli, L., & Benini, L. (2020). Fann-on-mcu: An open-source toolkit for energy-efficient neural network inference at the edge of the internet of things. *IEEE Internet of Things Journal*, *7*(5), 4403–4417.

Watson, R. A. (2014). Use of a machine learning algorithm to classify expertise: Analysis of hand motion patterns during a simulated surgical task. *Academic Medicine*, *89*(8), 1163–1167.

Wei, X., & Radu, V. (2019). Calibrating recurrent neural networks on smartphone inertial sensors for location tracking. *2019 International Conference on Indoor Positioning and Indoor Navigation (IPIN)*, 1–8.

Wong, D. L. T., Li, Y., John, D., Ho, W. K., & Heng, C.-H. (2022a). An energy efficient ecg ventricular ectopic beat classifier using binarized cnn for edge ai devices. *IEEE Transactions on Biomedical Circuits and Systems*, *16*(2), 222–232.

Wong, D. L. T., Li, Y., John, D., Ho, W. K., & Heng, C.-H. (2022b). Low complexity binarized 2d-cnn classifier for wearable edge ai devices. *IEEE Transactions on Biomedical Circuits and Systems*, *16*(5), 822–831.

Wu, H., Zhang, Y., Wang, J., Wang, W., Xian, J., Chen, J., Zou, X., & Mohapatra, P. (2020). Ioceansee: A novel scheme for ocean state estimation using 3d mobile convolutional neural network. *IEEE Access*, *8*, 153774–153786.

Wu, J., Li, F., Chen, Z., Pu, Y., & Zhan, M. (2019). A neural network-based ecg classification processor with exploitation of heartbeat similarity. *IEEE Access*, *7*, 172774–172782.

Xiong, L., Ling, X., Huang, X., Tang, H., Yuan, W., & Huang, W. (2020). A sparse connected long short-term memory with sharing weight for time series prediction. *IEEE Access*, *8*, 66856–66866.

Yang, Z., Zhang, S., Li, R., Li, C., Wang, M., Wang, D., & Zhang, M. (2021). Efficient resource-aware convolutional neural architecture search for edge computing with pareto-bayesian optimization. *Sensors*, *21*(2), 444.

Yao, S., Wang, T., Li, J., & Abdelzaher, T. (2019). Stardust: A deep learning serving system in iot: Demo abstract. *Proceedings of the 17th Conference on Embedded Networked Sensor Systems*, 402–403.

Yu, C., Chen, J., Hu, X., & Feng, W. (2022). A lightweight model of pose estimation based on densefusion. *2022 34th Chinese Control and Decision Conference (CCDC)*, 5688–5693.

Zebin, T., Scully, P. J., Peek, N., Casson, A. J., & Ozanyan, K. B. (2019). Design and implementation of a convolutional neural network on an edge computing smartphone for human activity recognition. *IEEE Access*, *7*, 133509–133520.

Zhao, S., Yang, J., & Sawan, M. (2021). Energy-efficient neural network for epileptic seizure prediction. *IEEE Transactions on Biomedical Engineering*, *69*(1), 401–411.

Zhao, Y., Afzal, S. S., Akbar, W., Rodriguez, O., Mo, F., Boyle, D., Adib, F., & Haddadi, H. (2022). Towards battery-free machine learning and inference in underwater environments. *Proceedings of the 23rd Annual International Workshop on Mobile Computing Systems and Applications*, 29–34.

Zheng, T., Ardolino, M., Bacchetti, A., & Perona, M. (2021). The applications of industry 4.0 technologies in manufacturing context: A systematic literature review. *International Journal of Production Research*, *59*(6), 1922–1954.

Zhong, G., Wang, L.-N., Ling, X., & Dong, J. (2016). An overview on data representation learning: From traditional feature learning to recent deep learning. *The Journal of Finance and Data Science*, *2*(4), 265–278.

Zhu, T., Kuang, L., Li, K., Zeng, J., Herrero, P., & Georgiou, P. (2021). Blood glucose prediction in type 1 diabetes using deep learning on the edge. *2021 IEEE International Symposium on Circuits and Systems (ISCAS)*, 1–5.

Zouridakis, P., & Dinakarrao, S. M. P. (2022). Performance-aware lightweight dynamic early-exit-based gait authentication. *2022 IEEE International Symposium on Circuits and Systems (ISCAS)*, 1–5.