UNIVERSITY OF TWENTE.

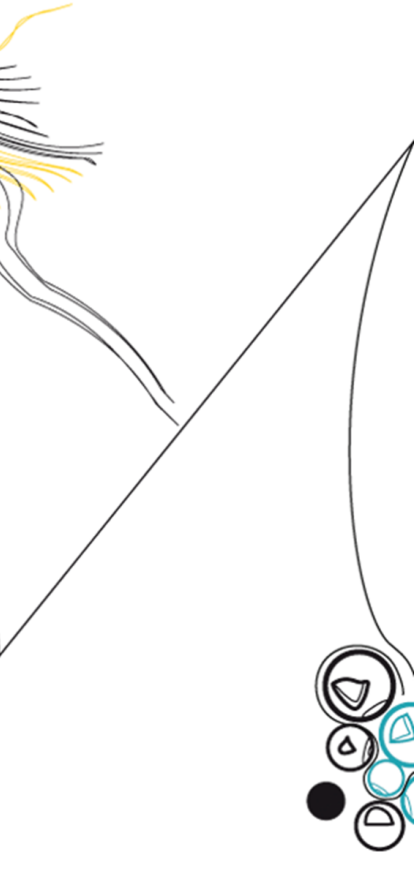
**Faculty of Electrical Engineering,
Mathematics & Computer Science**

AI in the Wild

Robust evaluation and optimized fine-tuning of machine learning algorithms deployed on the edge

A.P. van der Burgt

M.Sc. Thesis
June 2023



Supervisors:
prof. dr. P. J. M. Havinga
dr. J. W. Kamminga
ir. E. Molenkamp

Pervasive Systems Group
Faculty of Electrical Engineering,
Mathematics and Computer Science
University of Twente
7522 NB Enschede
The Netherlands

ABSTRACT

Estimates say that around 2.5 quintillion bytes of data are generated daily. Large computer clusters analyse this data in the cloud, requiring large amounts of energy for transmission and causing high latency. This research aims to resolve these issues by moving these computations towards the edge, which is called Edge Intelligence or TinyML. Little research has been done on EI or TinyML regarding evaluating deployed devices and further improving these devices. A system is often trained once and during deployment, not monitored and rarely actively maintained. This thesis shows the methods emplyed to evaluate deployed EI devices and offers methods to fine-tune deployed Machine Learning algorithms. Whilst deploying ML algorithms on edge devices, Out-of-Distribution data plays an enormous role. This OOD data is not gathered during training, but during deployment in a new environment. To research the influences of a new unknown environment, represented by OOD data, a dataset was made which contains six distributions, each containing ten objects.

To evaluate a deployed ML algorithm, it first should be determined how many images, including their predictions, should be sent to the cloud to identify the classification performance accurately. This research shows that 15 to 25 images per class are needed for every model, for a 95% confident Confidence Interval of 1% to 2%.

When deploying an EI device, images and predictions must be sent back to a central point for evaluation. Thus a power-efficient transmission protocol should be identified. This research shows that NB-IoT is the most power-efficient transmission protocol for sending image data.

ML algorithms can be used in the cloud and on edge devices. However, moving from cloud to edge devices and quantisation affects classification performance and power consumption. Complete and quantised models on the cloud have similar accuracy, but quantised models take longer to run inference. When comparing performance between quantised models in the cloud and on the edge CPU, there is a 7% to 17% accuracy difference. Finally, accuracy is similar when comparing edge CPU with edge TPU. Power consumption was reduced by 90% with the edge TPU due to decreased inference time. This research compared MobileNetv2, EfficientNetB0, EfficientNetV2B0, EfficientNetV2S and InceptionResNetV2. The first three models were more power efficient. This was due to decreased inference time of these models. The latter two were dropped in this thesis due to the unsuitability of the models.

When the performance of a deployed device is identified, fine-tuning can be used to increase the performance. This can be done in the cloud, and updated weights are sent to the edge device. When performing fine-tuning in the cloud, data may need to be sent from the device to the cloud and vice versa, which takes a lot of power. It is possible to perform fine-tuning on the device itself, but this can consume a lot of power. This research aims to identify where the trade-off lies in power consumption and final accuracy of deployed EI devices. The cloud-based fine-tuned models perform well on the cloud but suffer the same performance drop as the non-fine-tuned models when moving to the edge, with the accuracy difference varying from 10% to 25%. When using fine-tuning techniques on the edge, the overall classification performance is higher than when running cloud-fine-tuned algorithms on the edge, with an accuracy increase of about 10% to the best-performing cloud-based fine-tuning.

Considering the trade-off between cloud and edge fine-tuning, there is a crossing point where it is more power efficient to use cloud-based fine-tuning. In this research, for MobileNetV2, this crossing is located at around 480 images per class for fine-tuning. For EfficientNetV2B0, this is 8 images per class. For both holds that even when fine-tuning for 1 image per class on the edge, classification performance is higher than fine-tuning on the cloud. Power efficiency in its totality can be increased by using compressed images, as often the classes are still recognizable for humans. Lastly, insight is given into the different parameters which influence the battery life of a remotely evaluated fine-tuned EI device.

Overall, the presented work discusses a method for evaluation of a deployed Edge Intelligence device and provides insights into different methods of fine-tuning the Machine Learning algorithm deployed on the EI device. This thesis shows that it is possible and worth the trade-off between accuracy and power consumption to use edge hardware to fine-tune edge deployed ML algorithms on the edge device themselves, thus improving the classification performance for an edge deployed algorithm.

ACKNOWLEDGEMENTS

Dear reader, thank you for reading my master's thesis. Getting the thesis to where it stands now took me a while. Primarily due to constantly trying to squeeze out a bit more to be able to present an even better thesis. I found working on small microcontrollers and computers to be something that was most interesting to me during my studies. Placing it into context for it to be finally used in wildlife preservation was a definite bonus for me. In this way, a passion of mine could be used to help the world further to a better place.

I want to thank my supervisor Jacob Kamminga for a year of hard work and for being there to spar with about ideas. Furthermore, being there to answer the endless streams of questions that I asked and encountered.

Furthermore, I would like to thank my friends and family for being there when I needed it. Graduation work is not always easy, so talking to them and putting things in perspective was always nice. Specifically, I'd like to thank Maartje and Sander for the brave task of reading through my entire thesis for the first time to weed out the most obvious mistakes and imperfections. Likewise, Michael and Puck helped me to lift the thesis and my own writing to a higher level. Next to that, I would like to thank everyone who made my student life over the past seven years as to what it has been, something great for me to remember.

I hope the thesis that lies before you, is as interesting to you as it has been to me, and will inspire you to work further on the subject of this thesis, or in the topic of running machine learning on resource constrained devices.

CONTENTS

ACRONYMS

AE Autoencoder. 23
AI Artificial Intelligence. 6, 10, 14
AIoT Artificial Intelligence of Things. 14
ALSLR All Layers Small Learning Rate. 2, 5, 54–58, 60, 62, 63, 67, 70, 93, 96, 99
ANN Artificial Neural Networks. 17

CI Confidence Interval. ii, 3, 4, 35, 39, 41–43, 69, 82–87
CORAL Correlation Alignment. 22
CPU Central Processing Unit. ii, 2–5, 7, 8, 19, 50–53, 59, 60, 70, 88, 91

DANN domain-Adversarial Neural Networks. 22
DMM Digital Multimeter. 51
DSE Design Space Exploration. 35, 50
DT Decision Trees. 17

EI Edge Intelligence. ii, 2, 4, 6, 7, 9, 10, 14–17, 19–21, 24–26, 32, 42, 47, 48, 67–69
ERM Empirical Risk Minimization. 22

FL Federated Learning. 14, 18, 20, 21, 24, 25
FN False negative. 38
FP False positive. 38

GPU Graphics Processing Unit. 7, 19, 29, 44, 50, 52, 54

ID In-Distribution. 5, 22, 23, 32, 34, 94
IID Independent and Identically Distributed. 14, 18, 19, 24, 25
IoT Internet of Things. 6, 9, 14, 49

k-NN K-Nearest Neighbour. 17

LLRT Last Layer Randomized weights Training. 2, 5, 54, 55, 58, 60, 62, 63, 67, 71, 93, 96, 99
LLSLR Last Layer Small Learning Rate. 2, 5, 54, 55, 58, 60, 62, 63, 67, 70, 71, 93, 96, 99
LR Logistic Regression. 17
LSVM Linear Support Vector Machines. 17

MCU Microcontroller Unit. 14, 16, 17, 19, 24
MIPI CSI-2 MIPI Camera Serial Interface 2. 45
ML Machine Learning. ii, 2, 6–11, 13–21, 24, 25, 29, 35, 36, 38, 40–42, 50, 51, 53–55, 59, 66–69, 71, 72

OOD Out-of-Distribution. ii, 2, 3, 6–8, 21–25, 32, 34–36, 42, 43, 69–71, 95, 97

RF Random Forest. 17

SL Supervised Learning. 23, 25
SSL Self-Supervised Learning. 23
SVM Support Vector Machines. 17

TF lite Tensorflow Lite. 19, 29, 35, 40, 50, 52, 60, 70, 72
TinyML Machine Learning on Microcontrollers. ii, 6, 14–17, 19, 20, 24–26, 29
TN True negative. 38
TP True positive. 38
TPU Tensor Processing Unit. ii, 2–5, 7, 8, 29, 30, 35, 40, 45, 46, 50–54, 58–60, 70, 88, 92, 98

USB Universal Serial Bus. 45

WSL Windows Subsystem for Linus. 45

LIST OF FIGURES

LIST OF TABLES

1

INTRODUCTION

Estimates say that around 2.5 quintillion (10^{18}) bytes of data are generated every day [1, 2]. This is not only done by hand but more and more by autonomous sensors. These sensors include Internet of Things (IoT) devices placed in different environments and devices like smartphones [3]. These environments range from home environments for intelligent home applications to industry IoT devices and sensors that are connected and work together by gathering data to monitor devices and applications that are run.

Using dedicated computing clusters in the cloud to handle large amounts of data is gradually becoming unable to provide the needed computing power [4]. Next to this needed computing power, the power consumption by a large computing cluster is also large, and this can be reduced by performing calculations on the edge in advance [5, 6, 7]. Furthermore, high latency is introduced by sending data to the cloud and receiving the prediction [5, 6, 7, 8, 9]. By performing computations on the edge, this latency is reduced.

To counteract this problem, research has shifted to move these computations towards the edge [5, 6, 4, 10]. Moving these computations towards the edge allows for low-latency data handling and less transmission overhead. Edge Intelligence (EI) is the solution for this, as it combines the powerful computation of Artificial Intelligence (AI) and Machine Learning (ML) with low-power edge devices. EI can reduce the large amounts of raw data to data that only holds the needed information. Cloud and edge are not exclusive and can complement each other [4].

The usage of EI introduces numerous challenges and requirements which should be managed, and in this thesis, the following are addressed: (i.) The EI devices are to be placed in remote locations and should not be obtrusive. This leads to physical limitations, but also limitations in computational power, memory resources, and available power. (ii.) The devices and sensors may encounter degradation, meaning there should be a way to monitor their performance at a distance. (iii.) The devices are placed in different locations, meaning they all have different surroundings, thus Out-of-Distribution (OOD) data is obtained by all of them. (iv.) Edge devices have increasingly powerful processing units, sometimes also a ML accelerator. The difference between such a ML accelerator and the normal processing unit should be investigated, as it may lead to decreased power consumption. (v.) The deployed ML algorithm performance may vary if deployed on the edge or in the cloud. It is challenging to find the difference in performance without a practical example.

1.1 OPEN ISSUES

From the literature research done in Chapter 3, open issues were found which are to be tackled in this research. These are identified to be the following:

Post-Deployment Monitoring of deployed Edge Intelligence devices. The research that is available for EI and TinyML has mostly been aimed towards the training of the models that need to be run. When deployed, these models are not monitored and evaluated on their post-deployment performance [11]. Doing so may result in being able to improve the performance of ML models during deployment. When the device is deployed, the accuracy at that new place should be evaluated by capturing images and sending them, along with their predicted classes, to the cloud. When this is sent to the cloud, a human expert may need to identify the actual label for the image, and accuracy can be calculated. Sending these images and labels is costly due to resource constraints like power usage. Therefore we would like to send as few of them as possible without compromising the evaluation.

The Out-of-Distribution nature of real-world data. Data captured in the real world often contains changing backgrounds due to sensor degradation, changing seasons, or the sensor being placed in a different location [12, 13]. This leads to the captured data being data which has not been seen during training, thus being Out-of-Distribution. An example of OOD data can be seen in Figure 1.

Power consumption of a deployed Edge Intelligence device. The power consumption of a deployed EI device has not been monitored. Next to that, there is a difference in power consumption between running inference in the cloud and running inference on the edge. Many different transmission protocols are used, and there has not been an identification for an ideal EI deployed device.

Performance difference of different processing units in a Edge Intelligence device. There has been research into running quantised models on edge devices [14]. Mostly this research concerns the CPU and the GPU. However, a ML accelerator such as the TPU has not been discussed in this research but should be a good addition to these comparisons.

Performance difference of a deployed Edge Intelligence device versus the cloud for OOD data. Research has been done on the performance differences between edge and cloud systems [15, 16, 4]. Nevertheless, this is never done with real-world OOD data in mind.

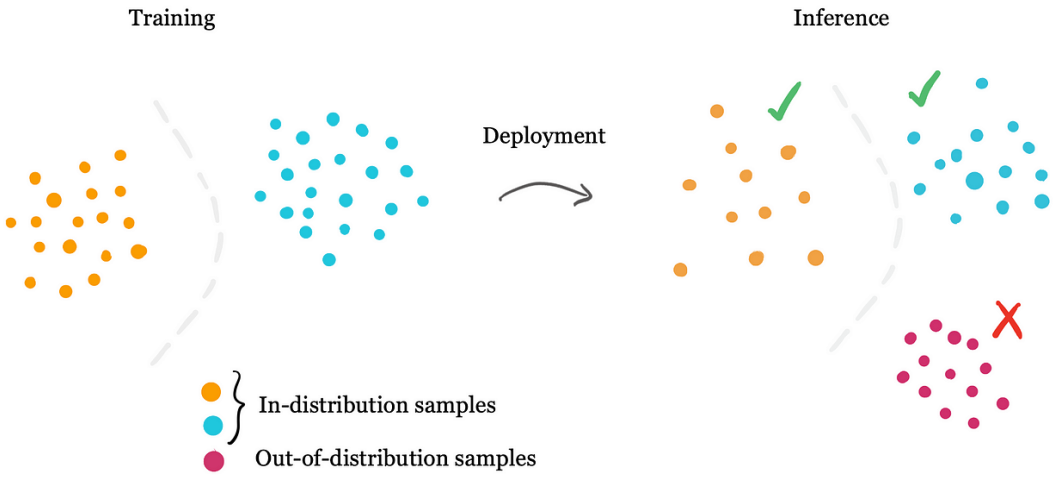


Figure 1: An example of how Out-of-Distribution data may occur, which would then be misclassified [17].

1.2 RESEARCH QUESTIONS

This research solves the above-mentioned research gaps and open issues. From these issues concerning OOD, real-life data and distribution shift in EI devices, the following main research question can be formulated:

> *How to achieve the best classification performance and reliable evaluation for a deployed edge intelligent device, while using the least amount of power?*

The first sub-question investigates the evaluation of a deployed edge intelligent device in a new distribution. The number of images that need to be sent back were minimized to minimise the power needed for evaluation. To find the minimum number of evaluation images, the first research question is posed:

> *How many images are needed, for a statistically reliable evaluation of a multi-class classification?*

The relationship between power usage and the remote evaluation of the deployed system needs to be identified. Deploying Edge Intelligence on resource constraint devices is a complex multivariate device with various trade-offs, including power efficiency, algorithm performance, and human effort (scalability). State-of-the-art does not show the proper evaluation of the effects of these factors. Therefore we pose the following sub-questions:

> *What is the trade-off between evaluation reliability, power usage, and human effort?*

> *What parts of remote evaluation influence power consumption?*

Next, running a Machine Learning model on an edge device needs adjustments to the model to fit the model on target hardware. The main techniques which are used are pruning, quantisation and knowledge distillation. These are all techniques which may affect the performance of the model. To investigate the effect of model quantisation, the following question is posed:

> *What is the trade-off between inference on edge CPU or edge TPU, and how does quantisation influence this?*

Fourthly, the effect of hardware architecture differences on the performance of a model is investigated. Various factors affect performance, such as different libraries for coding being used and running the code on other hardware. The following question is posed:

> *What is the trade-off between inference of cloud fine-tuned models on cloud and edge, and how does quantisation influence this?*

Related work uses fine-tuning to counteract the distribution shift [18, 19, 20]. Fine-tuning can be done by fine-tuning the quantised model on the device or by fine-tuning the full model in the cloud and then sending a quantised version to the edge device. These are two different ways of handling fine-tuning, yet they may affect classification performance and power consumption differently. Therefore, we investigate the following sub-question:

> *What is the trade-off between fine-tuning on the edge and in the cloud?*

To investigate all these questions, a dataset is needed, which consists of OOD data. This dataset is to be gathered and needs to be able to run on different models and edge devices. The dataset reflects the problem statement as accurately as possible. To ensure this, the dataset exists out of different domains to help train for OOD data. Furthermore, object variability is desired to ensure that classification is not too easy. Next, this dataset is to be easily extendable by others and easy to make, meaning that the classes and domains are available to anyone. This leads to the sub-question:

> *How to make a standardised edge AI distribution shift dataset, that allows supervision over different distributions?*

1.3 APPROACH

A real-world dataset is collected and annotated. The data comprises 10 classes in various environments to create separated distributions. This dataset is then used for the following experiments. To examine sub-question 1, the dataset is used to train and evaluate ML models, where the amount of evaluation data is incrementally enlarged. For each evaluation set, the performance difference between the full evaluation set and this evaluation set is determined. From this difference, a trade-off between reliable evaluation can be decided. To answer sub-question 2, a model is developed to visualise the power consumption of different transmission radios. To investigate sub-question 3, the ML models of sub-question 1 are quantised and compiled to run on the edge CPU and edge TPU. The classification performance and power consumption are then compared for a trade-off between the different architectures and models. Sub-question 4 investigates how cloud fine-tuned models perform on cloud architecture as compared to how they perform on edge architecture. The ML models, which could be compiled for the edge architecture, are compared based on classification performance. Lastly, sub-question 5 is examined. The trade-off between fine-tuning in the cloud and fine-tuning on the edge was investigated by deploying different ML models on a cloud infrastructure and an edge device with an edge TPU Machine Learning accelerator. This comparison was done based on classification performance and power consumption for the different fine-tuning methods.

1.4 THESIS ORGANIZATION

This thesis is structured in the following way. First, background research and a literature review are done in Chapters 2 and 3. Then an overview of the current state-of-the-art hardware and software used is given in Chapter 4. The gathered dataset is discussed in Chapter 5. Next, the experiments to answer the research questions are presented in Chapter 6 till 10. The conclusions drawn and work that can be done for improvement as found in the experiment chapters are concluded in Chapter 11

2

BACKGROUND

This chapter discusses the background of this research to be able to better understand the concepts and methods which are discussed later on in this research. Some general concepts in edge computing and Machine Learning (ML) are covered, to then be further extended in the next chapter with research by others. Furthermore, some methods are explored which can be used to reduce the size and complexity of a ML model. This can be used so that this ML model can run on an edge device with limited resources.

2.1 COMPUTING STRUCTURES

To better understand the types of computing structures that we are dealing with, it is best to identify the differences between the different structures.

2.1.1 *Cloud Computing*

Cloud Computing is the most used Internet of Things (IoT) data computation method as of now [21]. IoT in this context means that devices send their data to one central storage cluster to be stored or to be used in computations, thus storing data locally is unnecessary. Another advantage of cloud computing is that large computer clusters can perform calculations on the stored data. This often leads to fast calculations on large amounts of data. Another significant advantage is that multiple devices can access the data simultaneously. On the other hand, a disadvantage of cloud computing is that often high latency is introduced because of the larger communication overhead.

2.1.2 *Edge Computing*

Edge Computing takes data storage and processing completely away from the centralised perspective [21]. Data is stored and processed locally on the device that gathers the information, and only the results from data computation are sent to a cloud. This decentralized approach does make accessing the data harder. Edge computing does not depend on the cloud, and edge devices can operate independently. Nevertheless, a significant disadvantage is that edge computing hosts less powerful computing devices to use, leading to slower inference.

2.1.3 *Fog Computing*

Fog Computing can be seen as the layer between Cloud and Edge Computing [21]. It extends cloud computing by bringing data storage and computation closer to the edge. Fog Computing can distribute storage and computation over multiple nodes and store data physically closer to the edge. Nodes in the fog layer, the layer between cloud and edge, can decide whether to process it locally or send it to the cloud. While urgent requests are processed in the fog, the cloud handles less sensitive data or not-so-urgent requests. This indicates that it is more time-efficient than cloud computing, which may be necessary for some systems. However, a disadvantage would be the significantly larger overhead for communication.

2.2 EDGE INTELLIGENCE

Edge Intelligence (EI) combines edge computing and artificial intelligence. Different terminology has been used to describe how edge and end devices are used to process data, some pointing towards different goals than others. In Table 2 in Chapter 3, synonyms for EI can be seen. Edge Intelligence can be divided into different autonomy levels, as seen in Figure 2. Here EI, as discussed in this thesis, can be seen as levels 4, 5 and 6.

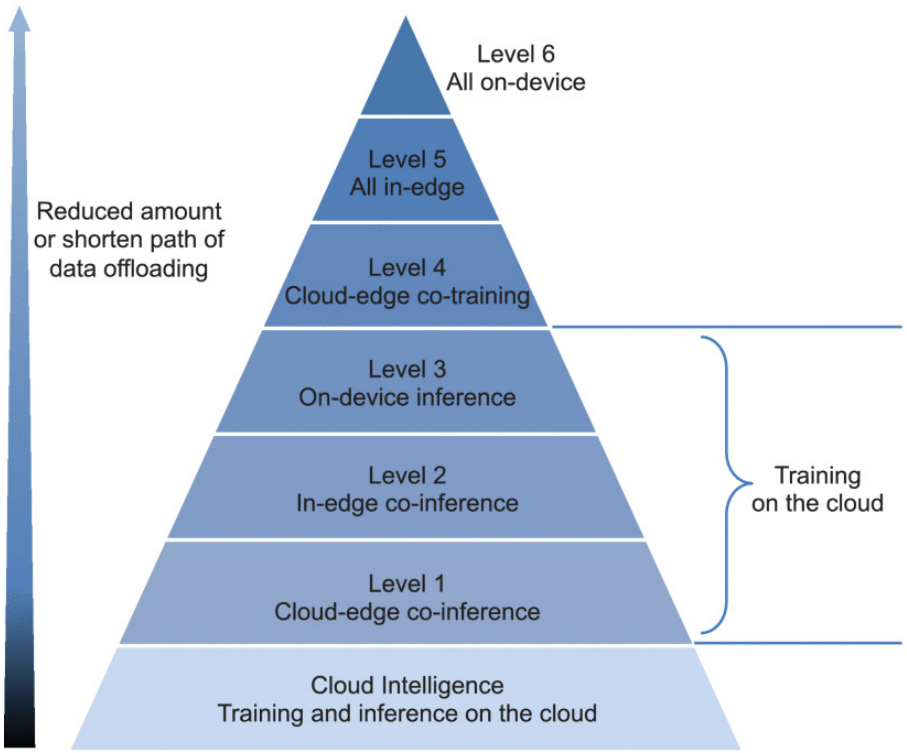


Figure 2: Levels of Edge Intelligence level 1 to 6 [10].

Figure 2 mostly shows where the training of the ML model happens and where the inference of new data happens. It is good to note the difference between edge and device, as the term edge can be the collaboration of multiple end devices, whereas the term device refers to just one device working on its own.

2.2.1 *Applications of Edge Intelligence*

To illustrate what EI can be, it is good to see where it is used in practice, including the AI techniques that can be used. The first application area where EI is present is Industry 4.0. This is accomplished by performing predictive maintenance [22, 23] or by optimising processes [3, 4]. For optimisation in the industry, EI is used to see where bottlenecks are located. For these applications, response latency, risk control, and privacy protection must be addressed [4]. Here EI can be used to process gathered data, which can then be evaluated using different frameworks (i.e. CEML) [24]. Moreover, EI is also used in the autonomous vehicle industry [4, 23, 25], where connectivity between such vehicles can vary from networking to sensing and the computation of the different sensors.

A second popular field is smart home and city [4]. This is due to the distribution of sensors and actuators, which can be used for indoor positioning systems or enabling robots for visual servicing. For a city network, EI is most interesting due to the geospatial nature of the data connected to the different sensor nodes [26].

Real-Time Video Analytics is an often applied field [4] enabling smart homes and cities. It is used in VR, AR and intelligent surveillance. EI is interesting in this application, as typically deep learning techniques are resource expensive and specifically, in surveillance, there must be low latency and high reliability. This computation can also be done on a hybrid level at different levels, as shown in Figure 2.

Within nature and agriculture [3, 10, 22, 27], EI can also be used. Application areas include using EI to detect diseases on plants and crops using a smartphone application. Since it runs on the end device, farmers in remote areas can also use the application. Next to that, ML powered nodes can be used to monitor wildlife, to ensure that humans do not interfere with them. Especially for anti-poaching applications [28, 29, 30], EI can be interesting due to minimally invasive monitoring and monitoring for extended periods. Another application in this area is nature conservation and Digital Species Identification. These systems can automatically detect species. They can be used to keep track of the population

in an area or to identify which prey is eaten by which predator. This allows for the mapping of the biodiversity in a location.

2.3 MACHINE LEARNING

A popular way of data analysis in recent years has been adding ML to existing applications. Therefore, knowing what steps ML consists of is important. This section briefly overviews the training steps and deploying a ML model. Figure 3 gives an overview of a typical Machine Learning model pipeline.

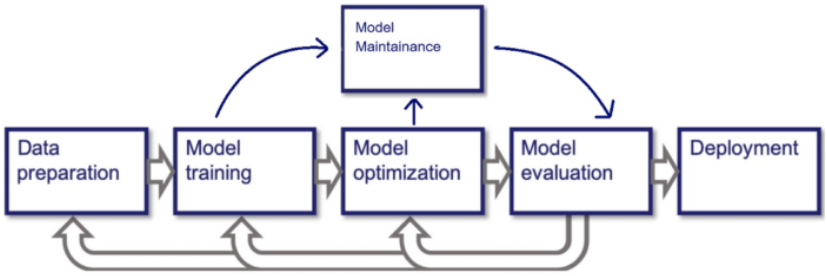


Figure 3: An overview of a Machine Learning model pipeline [31].

2.3.1 *Learning methods*

Machine learning can be implemented in various ways, of which the two most common methods are supervised learning and unsupervised learning. Supervised learning is an approach that uses labelled datasets to train. Often these algorithms are used for classification or regression as these are used to either classify or predict outcomes based on the inputs. Unsupervised learning is an approach that uses unlabeled datasets for training. Often these algorithms are used to find patterns in the data utilizing clustering or are used for dimensionality reduction.

Then there are some less often-used learning methods, such as semi-supervised, incremental, and active learning. Semi-supervised learning combines a small amount of labelled data with unlabelled data. With the help of the labelled data, it can cluster and label the unlabelled data and decide upon a decision boundary between different clusters. Incremental learning is a learning method in which the input data is continuously used to extend an existing model. For example, it could learn a new class in a classification algorithm. While some algorithms already inherently allow incremental learning, others must be adapted. Lastly, there is active learning, which is a part of supervised learning. Here a learning algorithm can ask a human expert or another information source like a larger ML model to label new input data so that the model can be updated.

2.3.2 *Model Training*

There are many different types of ML models (i.e. Decision Trees, Support Vector Machines, Naive Bayes and the now often-used Neural Networks), which all need to be trained before deployment [4]. There are different ways of training these models, which often depend on the final application for which the model is needed. When training a model, the data is split into parts: one to train the model (typically 60%), one to validate the model when training (20%) and a final part to test the fully trained model at the end (20%). The weights associated with the model are generally adjusted by an algorithm to increase or decrease a performance metric. The metrics which are often used are loss and accuracy. An example of the convergence of accuracy and loss during training can be seen in Figure 4.
When training, it is also possible to overfit or underfit a model [32]. Overfitting occurs when the model is trained on the training data for too long, and the performance on this training data increases, but the performance on the validation data decreases. This is often due to a lack of generalisation of the new

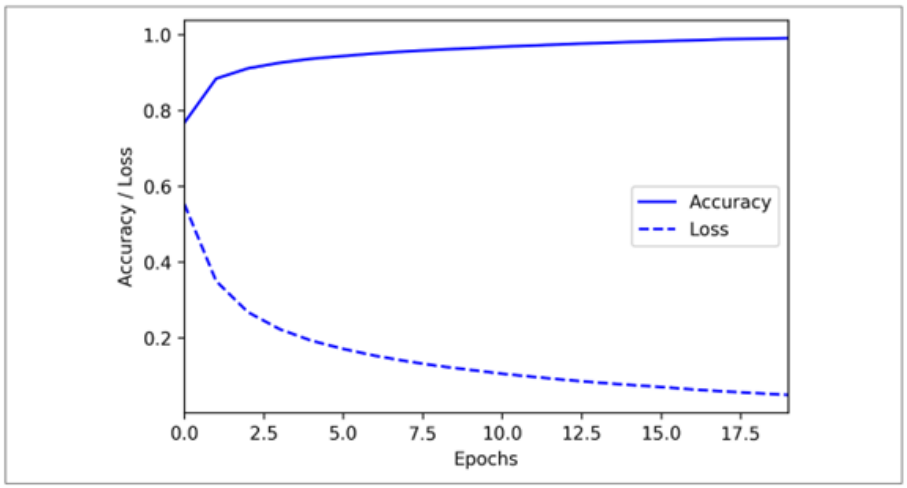


Figure 4: An example of accuracy and loss convergence during training [32].

data. Underfitting occurs when a model is not able to make good predictions. Often this happens due to the model being too small to capture the complexity of the data, or due to not enough training epochs. Figure 5 shows an example of underfitting, overfitting and a correct fit.

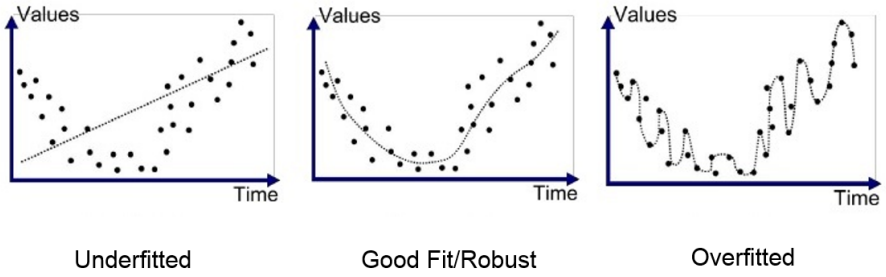


Figure 5: Fitting of models visualized [33].

2.3.3 *Model Validation*

When a model is trained, it is important to test the model between different training epochs to see how the model is performing [32]. This validation data is a better representative measure to see how the model is doing, as it is data that the model has not seen before. When comparing the training and validation performance, one can see if underfitting or overfitting occurs.

However, when the validation data is used, it may also occur that the model starts to overfit on both the training and validation data. To counter this, a final test dataset is used to test the fully trained model for its final performance. If the model returns a bad accuracy on the test set, then the model architecture may need to be improved, or another architecture should be chosen for the data. On the other hand, if the model performs well on the test set, it can be deployed to devices.

2.3.4 *Model Compression*

Bigger models often lead to better classification performance. Nonetheless, they also take valuable memory resources on the deployed device [23]. The trained model can be compressed to decrease the needed memory to store the model and possibly increase inference time. The six main techniques of model compression are shown and explained in Table 1.

Table 1: Model Compression Techniques

Model Compression Technique	Explanation
Model Pruning	Removing redundant weights or neurons, reducing memory footprint [23, 10].
Low-rank Approximation	Decomposing a large matrix with weights into two smaller matrices. This may lead to less computation complexity, thus faster results and a smaller footprint [23].
Parameter quantisation	Using a more compact data format to store parameters instead of a larger floating-point format. This reduces the memory footprint and increases energy efficiency [23, 10]
Vector quantisation	Grouping weights of a vector to share the same weight, thus compressing the model, reducing memory footprint [23].
Knowledge distillation	Training of a small neural network with the help of a larger one, reducing memory footprint [23].
Lightweight model design	Using specific tailor-made models for resource-constrained devices, reducing memory footprint [23].

2.3.5 *Model Inference*

When a ML model is deployed on a device, it can be used to infer new data, which is called inference [32]. For this to happen, the input is retrieved from sensors or other hardware, which may need to be filtered or pre-processed for use by the ML model. When inference is run, it is critical that the input is clean and there are no inconsistencies due to faulty sensors or other faulty hardware, as this could result in misclassification.

2.4 DISTRIBUTION SHIFT

A distribution shift is when the data with which a model works changes over time, influencing the predictions and causing the accuracy to drop [34]. There are two different distributions to identify, which are the source distribution and the target distribution. Respectively, these are the data on which the model is trained and the data on which the model runs inference after deployment. There are three different types of distribution shifts to distinguish, which are:

1. **Covariate shift.** This is when the distribution of the input changes, but the conditional probability of a label for a certain input remains the same.
2. **Label shift.** This is when the distribution of the output changes, but the conditional probability of a certain input for a certain output remains the same.
3. **Concept drift.** This is when the input distribution is the same, but the conditional probability of a certain output for a specific input changes.

2.4.1 *Detecting Distribution Shift*

Distributions shifts are not necessarily a problem, only when they cause your model to perform differently than desired [34]. It is good to investigate when this happens without an apparent reason. The model's performance can be checked in two ways: calculating accuracy or monitoring the distributions. Monitoring the distributions can be done for the input, label and conditional distributions. It is the easiest to keep track of the accuracy, but this does require ground truth labels, which are not always available. In the case when the ground truth labels are not available, observing the distributions is an option. In this case, the changes in the input distribution are often monitored. Examples include the distribution divergence of principal feature components, mean and variance of streaming inputs, or using the confidence score of predictions [35]. In addition, statistical tests can be used to detect if two distributions are significantly different.

3

LITERATURE REVIEW

In this chapter, the different terminologies surrounding Edge Intelligence (EI) are researched, and related works are investigated. Furthermore, the challenges with working on EI and TinyML are explored, and the open issues are identified.

3.1 METHODOLOGY

For this literature review, the sources used to search for related works are the University of Twente library [36] and Google Scholar [37]. The different keywords which are used are the following: *Edge Computing, TinyML, Edge Intelligence, Distributed Learning, Federated Learning, Edge Machine Learning, Edge Artificial Intelligence, Embedded Machine Learning, Embedded Artificial Intelligence, Artificial Intelligence of Things (AIoT), Embedded Intelligence, Post Deployment Monitoring*.

3.1.1 *Terminology*

As can be seen in the keywords in Section 3.1, similar terms exist, which are all related to one another but are not always synonyms. To get better insight, an overview has been made in Table 2. The main terminology that is utilized and investigated further in this research is EI and TinyML. This is because the most relevant papers were identified with these keywords. The search into the terminology as presented in Table 2 was not exhaustive, as can be seen by the lacking synonyms in the not further researched terminology.

Table 2: Difference in terminology between terms that were often associated to be Edge Intelligence. The terms of interest are highlighted.

Term	Explanation	Synonym
Edge Computing	Performing calculations on the edge, to offload the calculations from the cloud. This is often to reduce data transmission. Edge Computing does not necessarily include Artificial Intelligence.[4, 6]	n/a
Fog Computing	An architecture where the cloud is extended to be closer to the edge devices, thus improving latency and increasing security by performing calculations closer to the edge. [6] As opposed to cloud computing, instead of one central service, it offers multiple decentralised services.	n/a
Cloud Computing	A central fast and secure computing and data storage service that allows multiple devices to connect to it and use its computing resources [23]	n/a
Distributed Learning	Training ML models on multiple devices, using Independent and Identically Distributed (IID) data with high-performance nodes [23].	n/a
Federated Learning	Training ML models locally on data from multiple resource-constrained nodes, while preventing data leakage and using Non-IID data with varying amounts of data on each node.[25]	n/a
Edge Intelligence	The combination of Edge Computing and AI or ML [10]. An implementation of this could be that AI is combined with FPGAs or SoCs [8]. Next to that, AI implemented on IoT devices can also be considered Edge Intelligence [3].	Edge machine learning [6], Edge artificial intelligence [10], Embedded machine learning [8], Embedded artificial intelligence [38], Artificial Intelligence of Things (AIoT) [3].
TinyML	The deployment of ML on Microcontroller Unit (MCU) that are at the outermost edge of the infrastructure. Using Figure 2, it can be compared to level 3 and level 6 [15, 16, 32].	n/a
Embedded Intelligence	The ability for a device to contemplate its own performance. Although the term seems linked to EI, it is unrelated [39].	n/a

Figure 6 shows the overlap between the different terminology. The two most important parts of the terminology are Machine Learning (ML) and Edge Computing. However, Cloud Computing does not overlap with either of these, which is the exact opposite of Edge Computing. Fog Computing tries to marry these Computing architectures into one, bringing the cloud closer to the edge. Distributed learning and Federated Learning (FL) can be seen as an implementation of Fog Computing on the Edge Computing side, which is why they are shared terminology between these computing architectures. Embedded Intelli-

gence is a device's self-assessment, often with ML's help. This does not necessarily have to do with Edge Computing.

Finally, there is EI, where ML techniques are applied in the domain of Edge Computing. A more focused research direction within EI is TinyML, where everything is done on one device only. This is not necessarily the case with EI, as this also includes co-training options or pre-training.

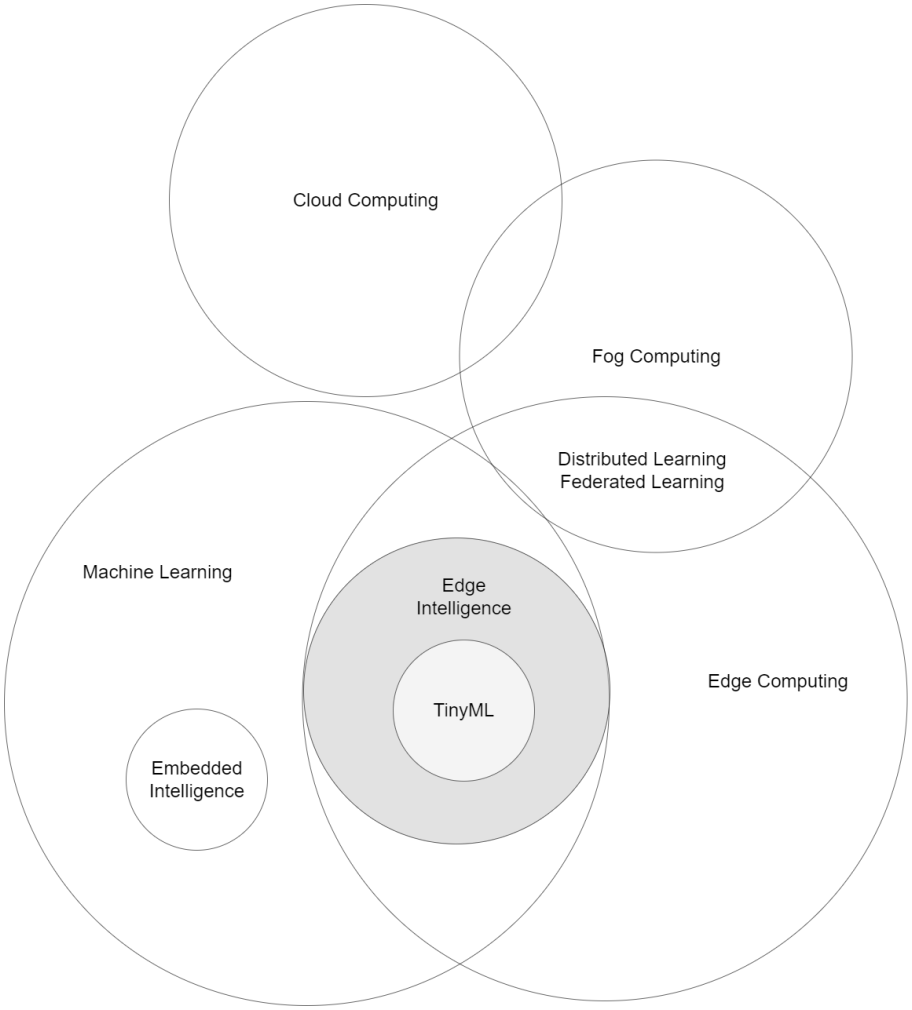


Figure 6: Venn Diagram showing the overlap between the different Terminologies

3.2 EDGE INTELLIGENCE

This section discusses the history of Edge Intelligence to give insight into where EI comes from. Furthermore, it outlines how other research has tackled the steps necessary to implement EI or TinyML.

3.2.1 *History of Edge Intelligence*

The first mention of the concept of Edge Intelligence (EI) can be found in the paper "The Case for VM-Based Cloudlets in Mobile Computing" [40], where "Cloudlets" are mentioned, which are smaller computing nodes close to the end-devices. These are what we would call edge nodes where computation could happen. In the whitepaper "A Berkeley View of Systems Challenges for AI" [41], the cooperation between cloud and edge is discussed, as Cloud-Edge systems are mentioned. The whitepaper covers the movement of placing cloud-located intelligence and systems on the edge to improve security, privacy, latency and safety. However, it mentions that extensive calculations are to be done by the cloud for applications such as drones, self-driving cars and home robots. These two papers are the first mentions of EI as defined in Table 2. Research into EI has been increasingly attractive in previous years due to an increase in large amounts of data gathering. Because of that, research into EI is relatively new. There are a lot of challenges to overcome as they do not have a defined solution, and open issues to still be solved.

3.2.2 *AI in Edge Intelligence*

Machine Learning is an often used way of data processing for Edge Intelligence. This section describes the difference in processes between ML in the cloud as compared to using ML on the edge and MCUs. Next to that, it also discusses different research that tried overcoming challenges connected to that part of Machine Learning and future work that they still have.

Leroux *et al.* [11] present a structure for implementing TinyML, which makes it an interesting work to start with. Their proposed structure can be seen in Figure 7. Here three major phases are presented, which are model, model updates and telemetry. The model phase consists of making the model by selecting a suitable model, compressing the model where needed and the quantisation of this model. Then comes the model updates phase, where the model is trained and validated. Finally, in the telemetry phase, monitoring and anomaly detection are handled, which helps to ensure the model is still performing accordingly. Besides that, when deployed for others, it may be possible to bill these others for the usage of the model.

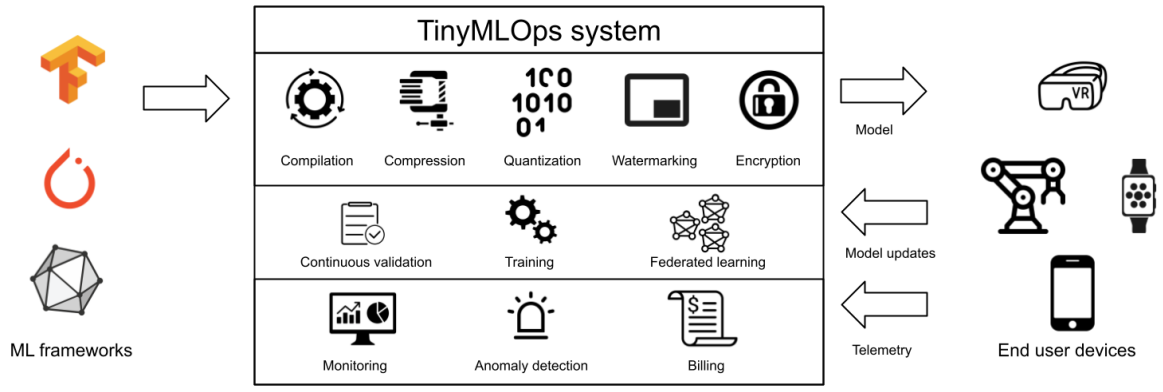


Figure 7: Overview of a TinyMLOps system [11]

3.2.2.1 *Model Selection for Edge Intelligence*

The first step of implementing Machine Learning on the edge consists of choosing an appropriate model to be run on the edge device. This must be a model which is small enough to fit on an edge device, or a model that can be downsized to fit on a resource-constrained device. Deng *et al.* [42] mention nodel

selection, and give an overview of appropriate hardware and the trade-offs that come with it, as one of the remaining challenges of AI on edge devices.

In the research by Sakr *et al.* [43], they attempted to check often-used ML algorithms on state-of-the-art edge devices. They investigated multiple questions, including the performance in accuracy and inference time, the effect of scaling data, and dimensionality reduction. To this end, they proposed a framework for TinyML training, inference and testing framework. Here, a computer was used to train and validate the models, and the inference was run on MCUs. The training computer was used to choose from the four models by tuning the hyperparameters and checking their final performance. These models are Artificial Neural Networks (ANN), Linear Support Vector Machines (LSVM), K-Nearest Neighbour (k-NN) and Decision Trees (DT). They trained these models for six different STM microcontrollers using ARM Cortex-M processors. They used six different datasets to validate their testing, which attempted binary classification, multi-class classification and regression. Their results showed that for ANNs and LSVMs, the accuracies were similar to the desktop implementation, but inference took longer. For k-NN models, it was found that the flash size of the microcontrollers was a limiting factor and a training set cap needed to be set, thus affecting accuracy. It performed similarly to ANN but did require a much larger footprint. DTs are the worst performer as they underperform compared to the rest of the models. However, it does have fast inference, which may be preferred in some cases. They conclude that their research would benefit from adding convolutional and recurrent neural networks. Next to that, more complex datasets such as images and audio streams may also help toward a better selection of models. Lastly, an analysis of unsupervised algorithms would be interesting in low-accessibility areas.

Huč *et al.* [44] try to test the robustness of five different ML models on a large imbalanced dataset. Their challenge itself is to look at anomaly detection to decrease the number of false alarms. The ML models they choose to compare against their Online Locally Weighted Projection Regression (OLWPR) are Logistic Regression (LR), Random Forest (RF), AdaBoost, Decision Trees (DT), Support Vector Machines (SVM) and Artificial Neural Networks (ANN). They split their datasets into smaller datasets with samples from anomalous classes and samples from computed clusters of the normal classes. After implementing the models on a regular computer, they concluded that the performance for the imbalanced datasets is mostly low for anomalous classes. For balanced datasets, performance is low for non-anomalous classes. Finally, they implement the models on a Raspberry Pi 4. Here they show that the Decision Trees model is the most promising solution due to its similar results for F1 score and low resource consumption, and fast inference. In future work, they mention using larger imbalanced datasets, pre-processing and optimization methods. Asides, they want to adopt several ML models to implement incremental learning in edge computing.

3.2.2.2 *Training for Edge Intelligence*

There are different ways of training a model for EI, which all are mainly concerned with where training is done. For example, training can be fully done by a central computing cluster, by the device itself or partly trained by the central cluster and then fine-tuned by the edge device, as can be seen in Figure 8.

Centralized models are run through big computing clusters [10, 23], which allow training and inference on the cloud. The centralized architecture can be seen in Figure 2 as Levels 1, 2 and 3. In decentralized architectures, the nodes train the models locally [10, 23]. This results in a more private way of storing and using information. However, the updated model parameters can still be shared from one node to another. This way of computing corresponds to level 5 in Figure 2. Hybrid architectures are a combination of centralized and decentralized architectures [10, 23]. Edge servers may train the model on the edge nodes, combine the other decentralized updates, or use centralized training with the cloud. This covers level 4 in Figure 2. The advantages and disadvantages of these architectures can be seen in Table 3.

Interesting research in the field of training for EI is De Prado *et al.* [13] looking into one of the significant challenges for automotive driving; the real-world environment changes over time. The problem they tried to solve was the difference in lighting. They achieved this by running a dual setup, employing a teacher-student model, to train a faster low-power system using a MCU, with a slower, more powerful system when needed. To this end, they made their own group of ML models to be used by the system, as well as their own dataset. They implemented a closed-loop learning system with two microcontrollers and cameras to achieve high accuracy on the inference microcontroller with low latency

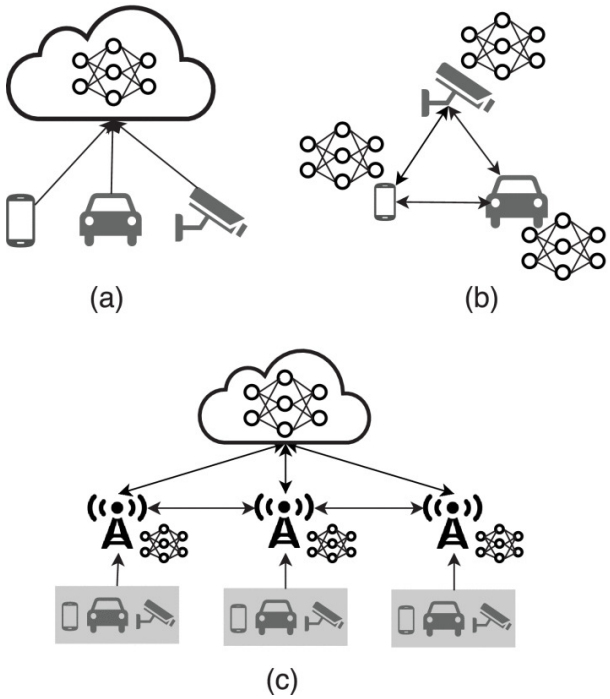


Figure 8: Architecture Modes of distributed learning. (a) Centralized, (b) Decentralized, (c) Hybrid [10]

Table 3: Training Architectures Advantages and Disadvantages

Architecture	Advantages	Disadvantages
Centralized	Utilize powerful computing clusters Fast training and inference	High Latency
Decentralized	Low latency Less data acquisition	Utilize less powerful computing clusters Slower inference
Hybrid	Medium latency Can use both types of computing clusters	More overhead for communication channels

and low energy usage. Future work can be found in performing training on-chip without needing a host (teacher) system. Besides that, it should also offer the ability to adapt continuously to the environment.

McMahan *et al.* [45] worked on extracting the benefits of shared models trained from shared data. They investigated this decentralized approach of multiple edge devices working together, which they named Federated Learning (FL). An improvement is that it helps reduce privacy and security risks. To uncover the exact workings of the shared models, they make a baseline model which uses Stochastic Gradient Descent (SGD), which they call FederatedSGD. Their coined approach is federated averaging, which improves upon FederatedSGD by, instead of having one parameter, having three. In their study, they also propose to have the edge devices do more computation, which may be problematic for a full-on edge approach. They try their work on different ML models, all using federated averaging to train high-quality models. Future work is identified as more research into differential privacy and multi-party computation.

Finally, in the research by Zhang *et al.* [46], they try to handle non-IID data with the help of FL. They mention FL as an excellent solution to the large amounts of data harvested and helpful in the privacy and security domain, as only the weights and biases are shared between the different edge devices. However, the data that one edge device may encounter may not represent the population distribution, thus being non-IID. They define weight divergence between clients as indicating a higher non-IID degree of data. Thus, the divergence of accuracy can be countered by selecting the nodes with lower non-IID degrees of data. The divergence of weights between clients is tested with IID and non-IID data,

where the weight divergence of IID data is lower than with non-IID data. The proposed model, Client Selected Federated Averaging (CSFedAvg), performs well, with fewer communication rounds, to achieve the desired accuracy. In addition, CSFedAvg improves training performance in the simulated scenarios, such as accuracy and convergence rate.

3.2.2.3 *Compressing a Machine Learning model for Edge Intelligence*

When a model is created and trained, it can be implemented on the target device. Often target devices in EI and TinyML do not have ample storage or computation capabilities, meaning the model should be compressed. Tensorflow Lite (TF lite) [47] is a popular service to do so, which is made to run on microcontrollers. Most often, a model is made in TensorFlow to be then converted to a TF lite model with the TF lite Converter [48]. Specific optimizations can also be chosen for conversion to reduce the overall size without compromising performance greatly. Another example of such a service is Intel's OpenVINO. They are more targeted towards CPUs and GPUs on higher-end machines. However, also have the availability to run on some edge devices like microcontrollers. OpenVINO also supports quantisation to ARM level CPUs, although this is less advanced than they do for others like CPUs and GPUs, which is expected, due to the restrictions of ARM CPUs. For OpenVINO, there is the post-training optimization tool, which can help further with quantisation.

Shamim [49] proposes a solution for an image recognition ML model on a resource-constrained embedded device. The framework used was TF lite for Microcontrollers, as developed by Google. MobileNetV2 was used to train the model by way of transfer learning. The training was done with the help of Edge Impulse Studio. The dataset has benign and (pre-)malignant tongue lesions to be classified by the model. To properly fit the dataset to a model for the MCU, Edge Impulse Studio can quantize the weights and biases of the default FLOAT32 precision into INT8 precision. This was found to influence the results slightly. The FLOAT32 model achieved a 99.8% accuracy with a loss of 0.002 on the validation set, with the INT8 model achieving similar performance with an accuracy of 98.42% and a 0.015 loss. Here the discrepancy in accuracy can be attributed to the quantisation from FLOAT32 to INT8. Unfortunately, the author provides no future work.

Next is the work from Mohan *et al.* [14], which attempted to detect people with and without facemasks. They set out to investigate the difference in accuracy the quantisation process brings from FLOAT32 models to INT8 models. The dataset used was a combined dataset of three different datasets and also augmented with OpenCV's interpolation methods, to increase the number of images. All images were resized to 32 x 32 as the authors found that this was the optimal size for the buffer of the OpenMV Cam. The accuracies of the SqueezeNet, Modified SqueezeNet and proposed model are respectively 98.53%, 98.99% and 99.83% for the INT8 model. Interestingly, the INT8 models outperformed the FLOAT32 models for every model. Furthermore, the modified SqueezeNet performed better than the unmodified model. This may indicate that smaller models can generalize better to new data. Furthermore, due to the slight difference in accuracy, they mention the quantisation to INT8 to be a high success. Finally, they say that future research can be done by experimenting with 6-bit, 4-bit and binarized neural networks.

Finally, in the work of Giordano *et al.* [50], an intelligent tag is made, to make a low-cost edge device that can track the usage of power tools. Their biggest challenge is the lifetime of the battery-operated device. A TinyML model is chosen to detect four different classes, transportation, idle drilling, wood drilling and metal drilling. The authors made the dataset which consisted of 280 minutes of three-axis accelerations with a sampling frequency of 800 Hz. A window of 448 samples is used to detect the different classifications. Here, FFT detects the transportation class due to its high peaks at lower frequencies. A small neural network was made for the other classes, which was quantised from FLOAT32 to INT8. The final result of the FLOAT32 model is an accuracy of 93.3%. The INT8 model yielded an accuracy of 90.6% for the same test set. Furthermore, the research concludes that the memory size of the quantised model is about three times smaller, where the amount of training cycles, latency and energy is all reduced three-fold. Future work is identified by the authors to be the generalization of position on a tool and extending the number of classes of interest.

3.2.2.4 *Inference for Edge Intelligence*

Inference, as discussed in 2.3.5, can be extended toward edge applications through solo-inference, hybrid co-inference and peer-to-peer co-inference [23].

Solo-Inference can be described as an inference model only running on one device, whether on the device itself, on the edge, or in the cloud, see Figure 9. The data is sent from the device to the relevant infrastructure for edge and cloud.

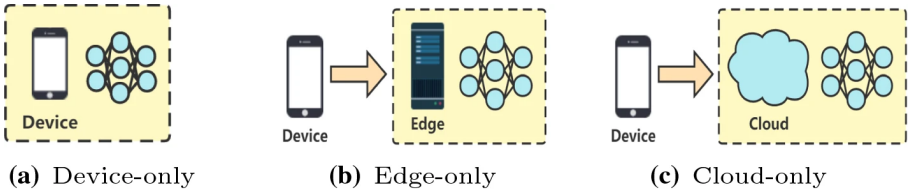


Figure 9: Different types of Solo-Inference [23]

The general definition for Hybrid Co-Inference is that inference runs on two or more infrastructures, as seen in Figure 10.

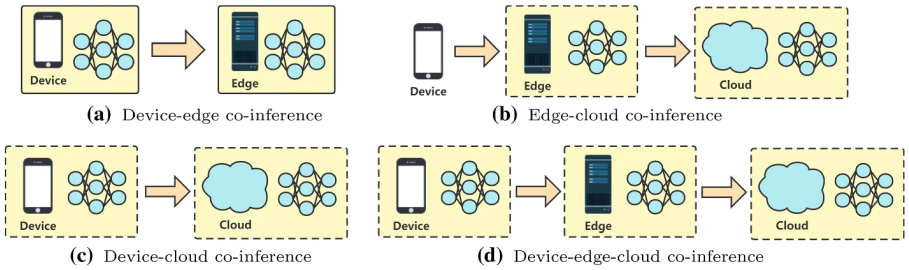


Figure 10: Different types of Hybrid Co-Inference [23]

Lastly, Peer-to-peer co-inference is very similar to hybrid co-inference. However, instead of dividing over multiple infrastructures, the inference is shared between the same infrastructures, as shown in Figure 11.

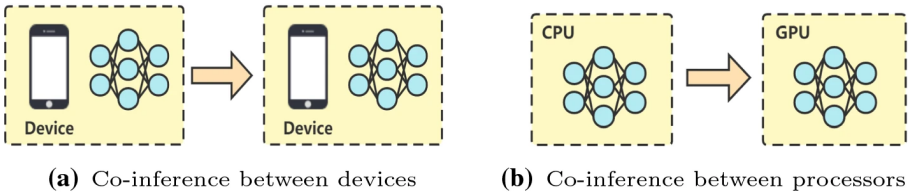


Figure 11: Different types of Peer-to-Peer Co-Inference [23]

Solo-inference is an often used inference type in EI and TinyML. As already shown in the research by Giordano *et al.* [50] and Mohan *et al.* [14], the entire training phase of a ML model is performed in the cloud, where the device only performs inference. Finally, in the research by Zhang *et al.* [46], the device itself often makes the inference. Updating and training can be shared and are done so in the context of FL. An example of Peer-to-peer co-inference can be seen in the works of De Prado *et al.* [13], where they used two microcontrollers. The more powerful system is only used when requested, leading to co-inference.

3.2.2.5 *Validation for Edge Intelligence*

The last step of running a ML model on edge devices is the same as running it on other systems; the model needs to be validated with a test set. The test set is often gathered from real-life data when the system is running. The validation of the edge systems is very similar to that of cloud systems. For example, McMahan *et al.* [45], who worked on the concept of FL, still uses performance metrics which are often also seen in more classic ML research papers. These graphs show the loss or accuracy against the number

of epochs. The only difference is that there is no single epoch with FL, but the global amount of epochs. Another example can be found in the research of De Prado *et al.* [13], who worked on the student-teacher model. They also use performance metrics such as accuracy, latency and energy consumption for validation. A third example comes from the research of Huč *et al.* [44], who made a classifier. They still use the terms Precision, Recall and F1 score, as is done in other research. Their results show confusion matrices better to exemplify their different ML models.

3.2.3 *Evaluation of Deployed Edge Devices*

Not much research has been done on evaluating deployed edge devices. This is probably because the evaluation of a deployed device is often complicated and cannot be done due to time constraints for most researchers. What makes it complicated is that to measure the performance, the ground truth is also needed to measure the performance. To do so, the sensor could be retrieved and known images or known variables can be entered to see if the output is a performance measure can be taken. However, this is not possible in traditional methods when the edge device is deployed and constrained in ways of processing power and, more importantly, energy consumption. The current state-of-the-art is to assume that over time the performance drops and the edge device need to be retrieved to do retraining as mentioned in Section 3.3.1

Only one mention of ML metrics to be used for post-deployment monitoring could be found, which is for a Stanford course by Chip Huyen [34]. She mentions a division of metrics to monitor, which are the raw inputs, features, predictions and accuracy. For all these monitoring possibilities, it is good to know that cumulative metrics may hide sudden dips in the metric, whereas a sliding window approach may not hide this. To measure the accuracy, ground truth labels are also needed. However, for others, this may not be necessary.

Huyen also gives an example of monitoring the predictions. This has to do with a possible distribution shift in predictions, which statistical tests can compute. If the weights and biases have not changed, a change in the output distribution often indicates a difference in the input distribution. However, it is good to note that this input distribution could be rightly shifted. For instance, if during the winter there are fewer birds because they are migratory birds, it is customary to see fewer predictions for birds. Furthermore, if the model predicts only one specific label, it could also indicate that something is malfunctioning.

Another possibility for post-deployment monitoring is to monitor the features. For example, metrics like the minimum, maximum and mean values can be computed for features, to see if they are in expected ranges. Furthermore, statistical tests can be used to check the underlying distribution of a feature or a set of features and to see if it has shifted. There are some significant drawbacks compared to input monitoring for feature monitoring. First, it is more computationally expensive. Secondly, a slight input distribution shift can cause substantial performance losses. However, an individual feature's changes might not hurt the model's performance in practice.

3.3 DISTRIBUTION SHIFT

A large part of the open research in Edge Intelligence (EI) concerns the diversity in real-world data. This envelops OOD data and distribution shifts as the main reason for the need of post-deployment monitoring. This section describes ways of dealing with distribution shift, and related work implementing such techniques.

3.3.1 *Tackling Distribution Shift*

A distribution shift can be identified, but the main concern remains to solve this shift. For this, three methods are often used in the industry [34]. The first way, which dominates current research and practical uses, is to train models on massive datasets. The assumption is that when there is a distribution shift, new data points are close to the trained distribution and thus, inference happens correctly.

The second approach, which is not studied often, uses unsupervised learning to adapt a trained model to a target distribution. Here, distributions are considered to correct the models' predictions.

Another way is to use domain invariant learning to learn data representations invariant to the changing distributions.

The third approach is to retrain the model using labelled data from the target distribution. Challenges posed here are the considerations of training on either the new data, both old and new data, or perhaps continuously training the model with new data. This last approach can also be called fine-tuning. However, this can be difficult to perform correctly, as deciding on what data to use can be challenging. For instance, if you should use data from the last 24 hours, if you should use all data from the previous retraining onward or from the point where the distributions diverge [19].

Other techniques, which are not widely adopted yet, but are proposed by the scientific community, all have to do with on-device training [35]. One such technique mentioned by Saha *et al.* is to update the bias instead of the weights. This could include only training the base classifiers with a significant impact on final accuracy. Another method which can be considered is Special Learning Techniques. This technique stores activation maps from past training data in a quantised form to be used as replay data. This should allow for learning from non-IID data.

3.3.2 *Similar Studies*

The research that has already been done into distribution shifts are discussed in this section. An excellent research to start with is the work from Koh *et al.* [51]. This research is the development of a benchmark for real-life distribution sets. They gathered ten datasets ranging from wildlife photos to cell and satellite imagery. These datasets concern domain generalization (iWildCam, Camelyon, OGB-MoIPCBA, RxRx1, GlobalWheat), Subpopulation shift (CivilComments) or both (FMoW, PovertyMap, Amazon, Py150). They try to address two problems: domain generalization and sub-population shift. Domain generalization concerns training on certain domains while testing on other domains. Sub-population shift focuses on training on all domains while testing on the same domains with a different proportion than the training. For evaluation, Koh *et al.* [51] propose different metrics for the different datasets. This is due to the nature of the data. Nevertheless, for all datasets, they evaluate the performance of In-Distribution (ID) and Out-of-Distribution (OOD) data. For domain generalization, they have CORAL [52] and IRM [53], and for subpopulation shift, they have Group DRO [54]. However, almost none of these latter models improved upon their baseline of Empirical Risk Minimization (ERM) models, except for the CORAL model for the iWildCam dataset and the Group DRO model for the CivilComments dataset.

Sagawa *et al.* [55] worked on extending the benchmarking tool presented by Koh *et al.* [51] for unsupervised domain adaptation. Here, some of the ten datasets used by Koh *et al.* were extended by unlabeled data, which can be used for training the models. Next to that, for the evaluation part, they sought three different methods: domain-invariant, self-training, and self-supervision. Concerning domain-invariant methods, they cover domain-Adversarial Neural Networks (DANN) and Correlation Alignment (CORAL). These models learn feature representations that are consistent across different domains. For self-training, they investigate the pseudo-Label, FixMatch and noisy student models. The aim of self-training is the concept that models label unlabeled examples, which they also use for training. Finally, there are self-supervision models consisting of SwAV and Masked Language Modeling (MLM). Self-supervision models learn representation by training on unlabeled data via other tasks. Like in the paper of Koh *et al.*, they compared the In-Distribution and Out-of-Distribution performance of the models on the datasets. The results show that for image detection ERM outperformed the newly posed models.

In the work of Wiles *et al.* [56], multiple models, which are considered to improve the robustness of models to distribution shift, are evaluated and compared. These improvements involve architecture choice, heuristic data augmentation, learned data augmentation, domain generalization, adaptive approaches and representation learning. These models are trained on six datasets (dSprites, MPI3D, SmallNorb, Shapes3D, Camelyon17, iWildCam) with three distribution shifts for these datasets. These distribution shifts included spurious correlation, low-data drift and unseen data shift. Spurious correlation makes a new data set for training with the correlated distribution and N samples from the uncorrelated distribution. Low-data drift means that we only see N samples of specific features, and for all other features, the model can access all features. An unseen data shift is a case of low-data drift, where N is set to 0. The results for the different models vary greatly for spurious correlation and low-data drift. For spurious correlation, CycleGAN performs consistently best, with pre-training on ImageNet also resulting in a boost in performance. For low data drift, pre-training on ImageNet performs consistently best, while

CycleGan, most domain adaptation methods and ImagNet augmentation also boost performance. Finally, Wiles *et al.* mention that the key takeaways are:

- Not one method always performs the best.
- Pre-training is a powerful tool across different shifts and datasets.
- Heuristic augmentation improves generalization if the augmentation describes a feature.
- Learned data augmentation is effective across different conditions and distribution shifts.
- Domain generalization algorithms offer little performance improvement.
- The precise attributes we consider directly impacts the results.

The work of Shi *et al.* [57] looks into the performances of pre-trained models on distribution shift. They investigate supervised learning and Self-Supervised Learning (SSL) and Autoencoder (AE) models. They primarily concern the SSL models, which enforce invariance between representations of two augmented views of the same image, and AEs, which learn by image reconstruction. Shi *et al.* worked on robustness to spurious correlation under synthetic and realistic distribution shifts. Respectively, these are based on synthetic datasets and more realistic scenarios of datasets. In this work, the work on WILDS [51, 55] is mentioned as the realistic distribution shift. The synthetic distribution shift concerns designed datasets with one simple feature (e.g. colour) and one complex feature (e.g. shape). Next to that, they test the retraining of the linear head on ID and OOD training sets and use the OOD data as a test set for both to measure the performance of the model. They trained eight learning models, which are SSL models (SimCLR, SimSiam, BYOL), AE models (Autoencoder, Variational Autoencoder (VAE), β-VAE, Importance Weighted Autoencoder (IWAE)) and one SL model (not defined). For synthetic distribution shift, they found a correlation between the complex and simple features learned by SSL and AE models, where SL only learns the more superficial features. They find that SL is no better than SSL or AE for realistic distribution shifts. However, they find that for OOD generalisation, the performance of all models can be increased by retraining the linear head on a small amount of OOD data. Finally, they note that AE and SSL consistently outperform supervised models and that the OOD generalization performance is significantly improved by retraining the linear head on a small amount of OOD data.

3.4 CHALLENGES

EI and TinyML come with challenges to overcome before widespread usage can be implemented. Some of these challenges are captured within the use of EI and TinyML, and others are found through the works of researchers. Most of these challenges have been solved by other research or are an underlying challenge that always exists when working with EI or resource-constrained edge devices.

3.4.1 *Resource Constraints*

Edge devices are often smaller devices which do not have large computational abilities, large memory and abundant energy to consume [23, 58]. Most of these resource constraints also influence one another. For instance, using a processor that can process data faster often consumes more energy. Often edge devices are battery-operated or use a solar panel to generate energy to be used. This limits the amount of on-time a device can have, thus limiting the time spent on data computation. This can further reduce such a device's computational ability through energy conservation. Besides the energy consumption, one of the most significant constraints is the size of a deployed ML model, due to the small memory size of these edge devices. These memory and energy constraints often cause other researchers to implement quantisation on their ML model to reduce the computational time and complexity, such that the edge device does not have a long on-time.

3.4.2 *Non-IID Data, Data Drift and Deployment of Edge devices*

Non-Independent and Identically Distributed (IID) data has to do with the fact that data is retrieved from one sensor or one type of sensor, which does not represent the entire population, thus not being identically distributed. There are ways to deal with this, of which an often chosen solution is Federated Learning (FL) [45, 46].

Besides a change in data due to non-IID data, there is data drift. This is when test data differs too much from the training data. An example of this would be a camera that captures the outdoors. Due to the seasons, the backdrop changes gradually. Another term for the non-IID data as is presented here is Out-of-Distribution (OOD) data. An extreme example of data drift would be that the model is trained for spring and summer seasons but tested on autumn and winter images. This violates the IID data assumption [12]. This data drift can be countered in numerous ways to reduce the impact on classification performance. Some of these are used during this thesis, but it is shown in related work that some of these can be used for edge devices, such as online learning through a student-teacher model [13]. Another example is fine-tuning of deployed models on newly gathered data [34].

Due to the data drift and Out-of-Distribution data, post-deployment monitoring is a challenge often overlooked. Transfer learning [5, 10, 16, 42], which includes distribution shift, encapsulates some of the challenges that OOD data also brings. For example, data is captured on different sensors and processed by different MCUs which may lead to a false inference. Some researchers consider these challenges, but they are not explicitly sought after [13, 49].

3.4.3 *Computational Limits*

Implementing models to work on edge devices is difficult due to the limits of the computational efficiency of the edge device. As can be seen in the state-of-the-art in Chapter 4, the processors and memory used are limited. Not only do most processors not have dedicated FLOAT32 computation hardware, there often is not enough memory on the device to work with these FLOAT32 numbers efficiently [14]. Besides these hardware limits of the device, there is also the energy limits, as it is beneficial to deploy as small as possible models on the device. For this, quantisation, as described in Section 3.2.2.3 [14, 49, 50], is an interesting aspect of deploying the models on edge devices.

3.5 OPEN ISSUES

From the literature reviewed in Sections 3.2 and 3.3, the following open issues are identified.

3.5.1 *Methods beside Supervised Learning and Federated Learning*

As mentioned in the introduction, Chapter 1, edge devices generate a lot of data. Sakr *et al.* [43] and Huč *et al.* [44] showed that methods beside Supervised Learning (SL) requires this data to be labelled to be able to use it, which is why other learning methods may be beneficial. These methods include unsupervised learning [43], active learning [43], incremental learning [44], or semi-supervised learning [44]. These methods are not researched well in related work, but there are some works which show the potential of the use of these methods in Edge Intelligence devices [57, 59].

3.5.2 *Non-Independent and Identically Distributed and Out-of-Distribution Data*

Recent literature [5, 23] showed that a heterogeneous environment is often present at the edge. One edge device gathering data, does not cover the entire population. Even when deploying numerous of these edge devices, the entire population is not covered due to real-world data being very diverse. This results in the fact that these edge device use non-IID or OOD data. This could be countered by letting the different edge devices which use the same model share their weights and biases, better known as FL [46]. Another solution would be student and teacher devices [13].

3.5.3 *Further Quantisation*

Mohan *et al.* showed that quantisation from FLOAT32 to INT8 can lead to more generalized models, leading to better classification performance. They refer to the optimization of further quantisation as future work, as further quantisation to 6-bit, 4-bit and binarised neural networks may not necessarily lead to more performance loss, but lead to faster inference times and less power usage.

3.5.4 *Post Deployment Monitoring*

Filho *et al.* [5] shows that when an edge device is deployed with a ML model, it is important to keep monitoring them. Due to data drift, distribution shift, or faulty sensors, the performance of an edge device may degrade. The post-deployment monitoring is mentioned in TinyMLOps as coined by Leroux [11], but research into this subject was not found in this literature review.

3.5.5 *Lack of Evaluation Platforms for Edge Intelligent Models*

The evaluation of deployed ML models is critical. Recent literature [16, 4] shows that there is a lack of baseline Edge Intelligence models. For cloud models, there is a suitable way of comparing the model's functioning by inputting a benchmarking dataset like ImageNet or COCO and seeing if the model performs better than other models. A set of baseline models is still lacking for EI and TinyML applications. Next to baseline models, literature [15, 16] mentions a suitable benchmark dataset is also missing. The availability of a benchmark dataset could improve further insights and comparisons between different ML algorithms.

3.6 CONCLUSION LITERATURE REVIEW

From the open issues and challenges, we can see that the handling of OOD data and the post-deployment monitoring of performance are large challenges still left. Often these two issues are overlooked or left to the sidelines when implementing an EI or TinyML solution to a problem, making it interesting to do more research into these issues.

4

STATE OF THE ART

This chapter outlines the technologies that exist now and are used in other research or currently used by companies. These technologies include the hardware used in state-of-the-art research for EI and TinyML, but also software frameworks. The hardware and software frameworks used for EI and TinyML can be very different, so they are separated. The hardware can be found in Table 4 and 6. The software frameworks can be found in Table 5 and 7.

Table 4: Edge Intelligence Hardware [4].
Abbreviations consist of Neural Processing Unit (NPU), Application Specific Integrated Circuit (ASIC), Artificial Intelligence Processing Unit (AI PU) and Digital Signal Processor (DSP)

Hardware	Processor	CPU Clock	Connectivity	Based on	Organisation
Snapdragon 8 Series [60]	Qualcomm Kryo CPU	2.99 GHz	-	NPU	Qualcomm
Kirin 600/900 Series [61]	2x Cortex-A76, 16-core Mali-G76 GPU	2.68 GHz	2G/3G/4G	NPU	HiSilicon
Ascend Series [62]	3D Cube	320 TFLOPS@FP16, 640 TOPS@INT8	-	ASIC	HiSilicon
Helio P60 [63]	Arm Cortex-A53, Arm Cortex-A73	2.0GHz	Bluetooth, FM Radio, GNSS, WiFi	GPU and AI PU	MediaTek
Turing GPUs	-	-	-	GPU	NVIDIA
TPU v4 [64]	1050 MHz	275 TFLOPS	-	ASIC	Google
Xeon D-2100 [65]	-	3.0 GHz	Ethernet	CPU	Intel
Exynos 9820 [66]	ARM Cortex-A75, ARM Cortex-A55 and ARM Mali™-G76 GPU	2.68 GHz	LTE-M	NPU and DSP	Samsung
GrAI VIP [67]	Dual-Core CPU		Camera Interface	GrAICore	GrAI Matter Labs

Table 5: Edge Intelligence Frameworks [4]

Framework	Hardware	Languages	Publicly available	Organisation
Baetyl v2 [68]	Ram 1GB minimum, CPU at least 1	GO	Yes	Baetyl
Azure IoT Edge [69]	Raspberry Pi 3 (64-bit quad-core ARMv8 CPU)	C#	Yes	Azure
EdgeX [70]	ARM 64-Bit	GO, C	Yes	EdgeX Foundry
NVIDIA EGX [71]	NVIDIA Jetson, NVIDIA GPUs	-	Unknown	NVIDIA
AWS IoT Greengrass [72]	NVIDIA devices, Specific Greengrass devices	-	Unknown	Amazon
Google Cloud IoT [73]	Edge TPU ASIC boards	C#, GO, Java, Node.js, PHP, Python, Ruby	Yes	Google
OpenVINO [74]	CPU, GPU, iGPU, VPU, ARM-CPUs	C, C++, Python	Yes	Intel

Table 6: TinyML Hardware [16]

Hardware	Processor	CPU Clock	Power consumption (Typical load main processor)	Flash	SRAM	Connectivity	Sensors/Connectors	Mentioned
Apollo3 [75]	32-bit ARM Cortex-M4F	48 MHz, 96 MHz with TurboSPOT	6 uA/MHz	1 MB	384 KB	BLE5, FTDI SPI,USB	Accelerometer, HM01B0 camera, MEMS microphone	[76]
STM32F Discovery [77]	32-bit ARM Cortex-M4 FPU Core	48 MHz	128 uA/MHz	1 MB	192 KB	LQFP100 I/O, USB	Accelerometer, microphone	[76, 43]
ST IoTDiscovery [78]	ARM Cortex M4	48 MHz	120 uA/MHz	1 MB, 64Mbit-Quad-SPI	128 KB	8.211b/g/n,NFC, 868/915 MHz, BLE 4.1, USB	Microphone, accelerometer, gyroscope, barometer, gesture detection, humidity, temperature	
ECM3532 AI Sensor NSP [79]	ARM Cortex M3, NXP CoolFlus 16bit DSP	100 MHz	13 uA/MHz	512 KB	256 KB	BLE 4.2, RF, USB	Pressure, temperature, gyroscope, accelerometer, temperature, humidity	
Arduino Nano 33 BLE Sense [80]	nRF52840	65 MHz	11 uA/MHz	1 MB	256 KB	UART,SPI,I2C,USB,BLE	microphone, gesture, IMU, light, proximity, barometer, temperature, humidity	[81]
OpenMV Cam H7 Plus [82]	ARM Cortex M7	480 MHz	300 uA/MHz	2 MB (internal)	1 MB + 32 MB SDRAM	I2C, USB, CAN, UART	5MP Camera at 50 FPS	[49, 14, 83]
Himax EW-I Plus [84]	32-bit ARC EM9D DSP with FPU Core	400 MHz	Not found	2 MB	2 MB	SPI2, I2C, UART, USB	VGA Camera 60 FPS, Accelerometer, Microphone	
Thunderboard Sense 2 [85]	EFR32 Mighty Gecko wireless SoC	38.4 MHz	63 uA/MHz	1 KB	256 KB	2.4 GHz, USB, SPI	Temperature, humidity, ambient light, pressure, air quality, microphone, hall-effect, UV	
Sony's Spresense [86]	ARM Cortex M4F 6 Core	156 MHZ	384 uA/MHz	8 MB	1.5 MB	SPI, I2C, UART, I2S, GNSS antenna	Microphone, Camera	[76]
TinyML Board [87]	Syntiant NDP101 NDP 32-bit ARM Cortex M0	48 MHz	11 uA/MHz	256 KB	32 KB	UART, I2C	Motion, Microphone	
Arduino Portenta H7 [88]	Arm Cortex M7 Arm Cortex M4 GPU	480 MHz, 240 MHz	300/120 uA/MHz	16 MB	8 MB SDRAM	WiFi, BLE, 10/100 Ethernet Phy, USB MIPI DSI, MPI D-PHY	Temperature, camera extension	
Raspberry Pi 4B [89]	64-bit ARM Cortex A72 quad core Broadcom BCM2711	1.5 GHz	360 uA/MHz	-	256 KB	WiFi, BLE, CSI, DSI, HDMI, USB, Ethernet	Temperature	
NVIDIA Jetson Nano [90]	Quad-core ARM A57 128-core Maxwell GPU	1.43 GHz	460 uA/MHz	4 GB	microSD	Ethernet, HDMI, USB, GPIO, I2C, I2S, SPI, UART	Camera (RPi)	
AI-deck 1.1 [91]	GAP8, ESP32	168 MHz	26 uA/MHz	1 MB	192 KB	WiFi, URT, SPI	Monochrome camera	
Pico4ML BLE [92]	Raspberry Pi RP2040 DSP dual core	133 MHz	180 uA/MHz	4 MB	264 KB	BLE, USB, I2C	Camera QVGA 60 FPS, Microphone, IMU	
MKR Vidor 4000 [93]	Intel Cyclone 10CL016 FPGA 32 bit ARM Cortex M0	48-200 MHz	500 uA/MHz	2 MB, 256 KB	32 KB, 8 MB SDRAM	SPI, I2C, UART, USB, MIPI, u-blox NINA-W102	-	
Nicla Sense ME [94]	ARM Cortex M4	64 MHz	120 uA/MHz	512 KB	64 KB	BLE4.2, SPI, USB, I2C	Accelerometer, gyroscope, pressure, geomagnetic, gas, temperature, humidity	[76]
CC1352P Launchpad [95]	CC1352R Wireless MCU LaunchPad	48 MHz	280 uA/MHz	352 KB	8 KB	UART, I2C, SSI, I2S, 868/915/433 MHz, BLE, Thread, ZigBee, 802.15.4, Sub-1 GHz Connectivity	Temperature	
ESP-EYE [96]	32 bit ESP32	240 MHz	40 uA/MHz	4 MB	8 MB PSRAM	WiFi, USB, SPI, I2C, UART, BLE	2MP camera	
GAP8 [97]	RISC-V, Hardware convolution engine	250 MHz (FC), 175 MHz (C), 22.65GOPs	26 uA/MHz	512 KB	80 KB, 8 MB SDRAM	Serial, SPI, I2C, I2S, CPI, Hyperbus, UART	Extension Camera	[13]
GAP9 [97]	RISC-V, Hardware convolution engine	400 MHz (FC), 150.8GOPs	6 uA/MHz	1.5 MB	128 KB, 2 MB External	Serial, SPI, I2C, I2S, CPI, Hyperbus, UART	Extension Camera	
Nordic Semi nRF52840 DK [98]	ARM Cortex M4	64 MHz	120 uA/MHz	192 KB	24 KB	BLE5, Bluetooth mesh, Thread, Zigbee, 802.15.4, ANT, 2.4 GHz, NFC, UART	-	[76, 50]
Nordic Semi Thingy:91 [99]	ARM Cortex M33 nRF9160 SiP	64 MHz	81 uA/MHz	1 MB	256 KB	UART, SPI, I2S, NB-IoT, LTE-M	Color, light, humidity, air quality, temperature, pressure	
FRDM-K64F [100]	ARM Cortex M4	120 MHz	120 uA/MHz	1 MB	256 KB	Ethernet, CAN, SPI, I2C, UART, I2S	Accelerometer, magnetometer	[76]
Coral Dev Board [101]	Quad Cortex-A53 Cortex-M4F GC7000 GPU Google Edge - TPU coprocessor	2.3 GHz / 64 MHz / 4 TOPS (int8)	117/90 uA/MHz 2 TOPS/W	8 GB	1 GB	Ethernet, WiFi, Bluetooth 4.2, audio, GPIO, I2C, UART, I2S	Camera, USB interface	
Coral USB Accelerator [101]	Google Edge TPU coprocessor	4 TOPS (int 8)	2 TOPS/W	N.A.	N.A.	N.A.	USB C	
Coral USB Mini [101]	Quad Cortex-A53 IMG PowerVR GE8300 GPU	2.3 GHz	117 uA/MHz 2 TOPS/W	8 GB	2 GB	WiFi, Bluetooth 5.0, audio, GPIO, I2C, UART, I2S	Camera, USB interface, microphone	
Coral USB Micro [101]	Cortex-M7 Cortex-M4 Coral Edge TPU coprocessor	480 MHz / 240 MHz / 4 TOPS (int8)	300/90 uA/MHz 2 TOPS/W	1 GB	512 MB	GPIO, I2C, UART, I2S On-board camera and microphone	Ethernet, WiFi, Bluetooth 4.2 (not integrated)	
Maix Go [102]	Dual-core 64bit RISC-V	400MHz	0.83 TOPS/W	17 MB	8 MB	Camera, GPIO, OTP, UART, SPI, I2S		
OAK-D-Lite [103]	Robotics Vision Core 2	4 TOPS	0.35TOPS/W	N.A.	N.A.	IMX214 camera, OV7251 camera		

Table 7: TinyML Software Frameworks [15, 16, 104, 105]

Framework	Algorithms	Platforms	Languages	interoperable External Libraries	Publicly available	Organisation
TensorFlow Lite [47]	Neural Networks	ARM Cortex-M	C++ 11	TensorFlow	Yes	Google
TensorFlow Lite for Microcontrollers [106]	Neural Networks	Arduino Nano 33 BLE Sense SparkFun Edge STM32F746 Discovery kit ESP32 ESP-EYE Spresense	C++	TensorFlow	Yes	Google
Edge Impulse [8]	Neural Networks	Arduino platforms ESP32 Nordic platforms Spresense Syntiant Tiny ML Board CC1352P LaunchPad Raspberry Pi 4 Jetson Nano	Node.js, Python, Go, C++	-	Yes	Edge Impulse
PyTorch Mobile [107]	Neural Networks	Android IOS	Python	-	Yes	PyTorch
Embedded Learning Library (ELL) [108]	Neural Networks	ARM Cortex-M ARM Cortex-A Arduino micro:bit	C, C++	CNTK Darknet ONNX	Yes	Microsoft
ARM-NN [109]	Neural Networks	ARM Cortex-A ARM Mali Graphics Processors ARM Ethos Processor	C	TensorFlow Caffe ONNX	Yes	ARM
CMSIS-NN [110]	Neural Networks	ARM Cortex-M	C++	TensorFlow Caffe PyTorch	Yes	ARM
STM 32 Cube.AI [111]	Neural Networks	STM32	C	Keras TensorFlow Lite Caffe ConvNetJs Lasagne	Yes	STMicroelectronics
AIfes [112]	Neural Networks	Windows (DLL) Raspberry Pi Arduino ATMega32U4 STM32 F4 Series ARM Cortex-M4	C	TensorFlow Keras	No	Fraunhofer IMS
NanoEdge AI Studio [113]	Unsupervised Learning	ARM Cortex-M	C	-	No	Cartesiam
MicroMLGen [114]	SVM, RVM	Arduino ESP32 ESP8266	C	Scikit-learn	Yes	Particular developer
sklearnporter [115]	SVM Decision tree Random Forest Ada Boost Classifier k-NN Naive Bayes Neural Networks	Multiple constrained & non-constrained platforms	C, GO, Java, JavaScript, PHP, Ruby	Scikit-learn	Yes	Particular developer
m2cgen [116]	Linear Regression Logistic Regression Neural Networks SVM Decision tree Random Forest LGBM Classifier	Multiple constrained & non-constrained platforms	C, C#, Dart, GO, Java, JavaScript, PHP, PowerShell, Python, R, Visual Basic	Scikit-learn	Yes	Particular developer
weka-porter [117]	Decision trees	Multiple constrained & non-constrained platforms	C, Java, JavaScript	Weka	Yes	Particular developer
EmbML [118]	Decision trees Neural networks SVM	Arduino Teensy	C++	Scikit-learn Weka	No	Research Group
emlearn [119]	Decision trees Neural networks Naive Gaussian Bayes Random forest	AVR Atmega ESP8266 Linux	C	Keras Scikit-learn	Yes	Particular developer
uTensor [120]	Neural Networks	mBed boards	C++11	TensorFlow	Yes	Particular developer
TinyMLgen [121]	Neural Networks	ARM Cortex-M ESP32	C	TensorFlow Lite	Yes	Particular developer
CMix-NN [122]	Neural Networks	ARM Cortext-M	C	Mobilenet	Yes	Research Group
FANN-on-MCU [123]	Neural Networks	ARM Cortex-M PULP	C	FANN	Yes	Research Group

4.1 DISCUSSION OF SOFTWARE PLATFORMS

For software platforms, there are various choices which all have advantages and disadvantages. The platforms that were available and mentioned in related work are Edge Impulse [8], OpenVINO [74], OpenMV [82] and Neuton AI [124]. There are some major disadvantages for the latter three. Firstly, OpenVINO has a lack of support for Microcontroller hardware. Secondly, OpenMV only supports its own microcontroller and no others, which is also too restrictive. Thirdly, Neuton looks promising, but does not yet support image, video and audio.

Edge Impulse is the best documented, often used and most open platform for TinyML. It supports all types of devices from single board computers like the Raspberry PI to GPUs to microcontroller class hardware and TPUs [125]. Next to that, it also has data gathering, pre-processing and annotation tools. Furthermore, it supports some predefined ML models out of the box and offers so-called "expert mode" [126]. This allows a developer to create their model in TensorFlow combined with Keras.

Lastly, it also employs compilers to compile it to their "fully-supported" hardware and compilers to compile it to a generic Tensorflow Lite model, which can then be loaded onto your hardware, such as a Google Coral TPU Development Board [127]. Edge Impulse seems to be the preferred tool. However, they have restricted the open access to only allow 4GB or 4 hours of data, with no option for research purposes. Therefore, Edge Impulse is not suitable, as in this research, more data storage needs to be available to train and test the different ML models.

4.2 DISCUSSION OF EDGE HARDWARE

The choice of hardware could decide what models are used but also in what way data needs to be gathered and what software the models run. For this, different metrics were used, which are found in Table 8. These consist of compatibility of hardware for ML, the platform's ease of use, the size of the community, the availability of the platform, the fact that the platform can use a camera, microphone or both, and the total power consumption of the platform. Some of these metrics are more important in this research, and get a higher weight.

Table 8: Design Space Exploration of Hardware Choice, from "- -" to "++", which represents a score from -2 to +2. This score is multiplied with the weight and finally summed to a total which is shown in the last column.

Hardware	Capability of ML	Ease of use	Size of community	Accessability	Sensors of interest & Easy attachability	Power Consumption	Total
Weights	**3**	**1**	**1**	**2**	**1**	**2**	
Coral Dev Board [101]	++	+	+	++	+	+	15
Coral USB Mini [101]	++	+	+	+	+	+	13
Coral USB Micro [101]	++	+	+	+/-	++	+	12
Arduino Nano 33 BLE Sense [80]	+	+	+	+	+	++	12
Raspberry Pi 4B [89]	++	++	++	++	+	- -	11
Coral USB Accelerator [101]	++	+	+	+	- -	+	10
NVIDIA Jetson Nano [90]	++	++	++	+	+	- -	9
Apollo3 [75]	+/-	+	-	+	++	++	8
Arduino Portenta H7 [88]	++	+	+/-	+	+/-	-	7
Nordic Semi nRF52840 DK [98]	++	+	+/-	+	+/-	-	7
OAK-D-Lite [103]	++	+	+/-	+	++	- -	7
ECM3532 AI Sensor NSP [79]	+	+/-	-	-	++	++	6
Sony's Spresense [86]	++	+/-	+	+	+/-	- -	5
Nicla Sense ME [94]	++	+/-	+/-	+	- -	-	4
GAP8 [97]	++	+/-	+/-	- -	+/-	+	4
GAP9 [97]	+	+/-	+/-	- -	+/-	++	3
ESP-EYE [96]	+	+	-	- -	++	+	3
Maix Go [102]	++	-	-	-	++	-	2
OpenMV Cam H7 Plus [82]	++	+	+/-	-	+	- -	2
AI-deck 1.1 [91]	+	-	- -	+/-	+/-	+	2
Nordic Semi Thingy:91 [99]	+	+	+/-	+/-	- -	+/-	2
FRDM-K64F [100]	++	+/-	-	-	+	-	2
TinyML Board [87]	+	+/-	- -	-	-	+	0
Pico4ML BLE [92]	+/-	+	-	+/-	++	-	0
Himax EW-I Plus [84]	+	+/-	+/-	-	++	- -	-1
Thunderboard Sense 2 [85]	+	+/-	-	-	-	+/-	-1
ST IoTDiscovery [78]	-	+/-	- -	+	+/-	-	-5
STM32F Discovery [77]	-	+/-	- -	+	-	-	-6
CC1352P Launchpad [95]	+/-	+/-	-	-	-	- -	-8
MKR Vidor 4000 [93]	- -	+/-	+	+/-	+/-	- -	-9

Table 8 was then further refined. For this, the boards with a score of 9 and above were chosen. This selection has been further examined and presented in Table 9. As seen in this table, the Google Coral edge TPU boards are well above the rest. The final choice came to the Google Coral TPU Development Board, due to the available connection types, interface and wider availability.

Table 9: Design Space Exploration of Hardware Choice, from "- -" to "++", which represents a score from -2 to +2. This score is multiplied with the weight and finally summed to a total which is shown in the last column.

Architecture	Hardwares	Max Size for ML models	Performance (FPS for MobileNet-V2)	Ease of use	Power Consumption	Total
Weights	**N.A.**	**2**	**3**	**2**	**3**	
Coral TPU	Coral Dev Board [101] Coral USB Mini [101] Coral USB Micro [101]	+	++	+	+	13
Quad-core ARM A57 128-core Maxwell GPU	NVIDIA Jetson Nano [90]	++	+	++	- -	5
Quad Core ARM-A72	Raspberry Pi 4B [89]	+	+/-	++	- -	0
Nordic nRF52840	Arduino Nano 33 BLE Sense [80] Nordic Semi nRF52840 DK [98]	- -	- -	+	++	-2

5 STANDARDISED EDGE AI DISTRIBUTION SHIFT DATASET

In order to study post-deployment monitoring of EI devices and the impact of real-world OOD data, we collected a high-quality dataset with images of objects in different surroundings. In this chapter, the collection methodology, as well as the labelling process, of this image dataset is described.

Image datasets like CIFAR10 [128] and ImageNet [129] are used in many of the works presented in the literature review, to look at the influence of different surroundings in an image on the accuracy. However, these datasets are missing insight into which distribution objects are placed. Thus for this thesis, a dataset is created where objects are placed in different surroundings to control the distributions.

This dataset was used to study how different distributions can be used to generalise to distributions not seen in training. Several distributions, or environments, were identified and chosen where the objects are placed. To make a correct dataset for OOD data, this dataset was to have distributions that are distinct and not too similar. Furthermore, these surroundings should occur everywhere in the world, as the dataset needs to be able to be extended by others.

It is preferred that these objects are easily identifiable and easily obtainable by anyone to extend the dataset. Furthermore, having multiple variations, such as size and colour, of every object makes sure that the classification is not too easy, as otherwise, the models would perform too well in the different surroundings, defeating the purpose of the research into OOD data. Besides that, it would be helpful if the objects which are chosen overlap with already existing datasets. This makes it easier to use a model which is pre-trained on such a dataset, allowing for faster training and making transfer learning more accessible.

Furthermore, the number of images must be determined per class per distribution. This should be a reasonably high number to allow the model to converge during training.

5.1 METHODOLOGY

This section explains the way how the Out-of-Distribution (OOD) dataset is collected and processed, to be used in the remainder of this thesis. The classes of this dataset can also be found in the COCO dataset [130], to be able to use images from this dataset to make transfer learning easier. The distributions and classes are shown in Table 10.

5.1.1 *Data Acquisition*

High-quality images were taken, because images can always be downscaled but cannot be reliably upscaled. The photos were taken with a Lumix DMC-G80, with a 3840 by 2160 pixels resolution. 100 images were taken per class per domain. Photos were taken from different angles with different compositions. To simplify the data annotation process, the images were first grouped per distribution, and then per class. This was to ensure that In-Distribution (ID) and OOD testing can be done and labels can be easily retrieved. Examples of the images can be seen in Figure 12.

Table 10: The different distributions and classes in the dataset, with a description.

	Description
Distributions	
Cityscape	The object is surrounded by buildings and streets.
Forest	The object is surrounded by trees, moss and shrubs.
Office	The object is placed in a light room, with not much clutter, generally has some desks or PC equipment.
Park	The object is surrounded by nature. Differs from the Forest by having paths and open grassy places.
Pub	The object is placed in a cluttered, darker room. Often the furniture is also quite dark.
Uniform	The object is placed in front of a single colour background indoors. This is done indoors to make the lighting as uniform as possible.
Classes	
Apple	A piece of fruit.
Backpack	A backpack that is not worn. This does not include purses.
Ball	A ball used in different sports.
Book	A book of varying thicknesses and different colours for the cover.
Bottle	A plastic or glass bottle.
Fork	A piece of cutlery. Can be made of different materials.
Phone	A handheld smartphone.
Remote	Remote controls for appliances.
Scissors	A pair of scissors, with different shapes, colours, opened or closed.
Teddy Bear	A stuffed animal toy, of different shapes with different colours.

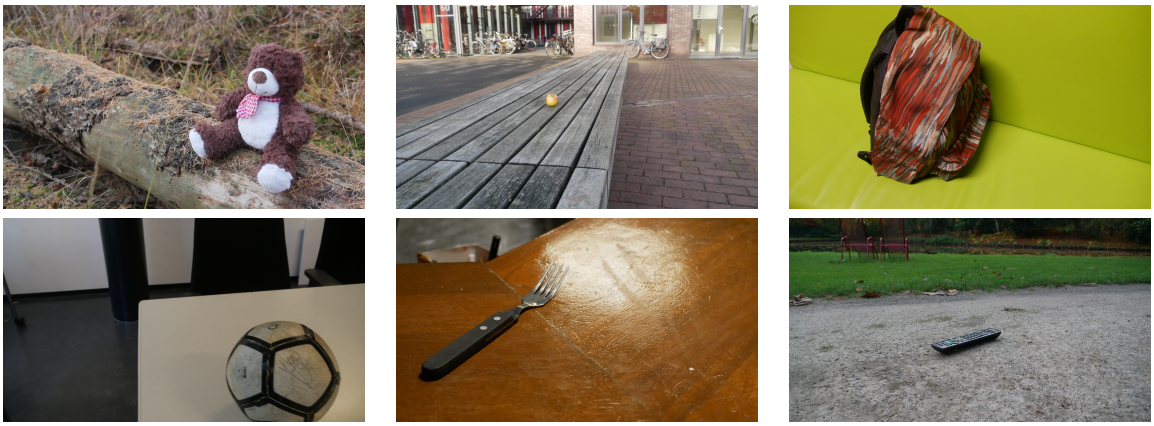


Figure 12: Example images of the created dataset. In respective order, bear in forest, apple in cityscape, backpack in uniform, ball in office, fork in pub and remote in park.

5.1.2 *Dataset Evaluation*

The quality of the dataset has been checked with the balance score [27] and the openness factor [131]. The balance score is a normalized value between 0 and 1, where 1 denotes a balanced dataset, and 0 is an imbalanced dataset. The dataset, which is presented here, is balanced because every class has the same size, thus the balance score of the dataset is 1.

$$BalanceScore = \frac{H}{\log k} = \frac{-\sum_{i=1}^{k}(\frac{c_i}{n}\log\frac{c_i}{n})}{logk} \tag{5.1}$$

As coined by [27], where:

H = Shannon entropy
k = the number of classes
n = the total number of data samples
c_i = the size of class i

The openness factor is given by Scheirer *et al.* [131] to indicate how challenging it is to classify a dataset. It is an estimate for OOD data detection difficulty and is determined by the number of classes which can be considered ID and which classes can be considered OOD. The openness factor varies between 0 and 1, where 0 represents a fully closed problem and larger values more open problems. This openness factor can thus give good insights into how OOD a certain problem is. However, the openness factor for real-world data can be difficult to determine, as there is a lot of variability in the data as discussed in Section 3.3 and thus a large number of possible OOD data in the real-world data.

As this research entails a multi-class classification, the openness factor should be 0 [131]. However, because different domains are used, the combination of an object and a domain can be considered a separate class. Therefore, the ID distribution consists of 5 domains with 10 objects, thus 50 classes. For the amount of OOD data, 10 classes are added to the 50 classes, leading to 60 testing classes. This would lead to a final openness factor of approximately 0.05, meaning that the problem addressed cannot be compared to an open set problem.

$$OpennessFactor = 1 - \sqrt{\frac{2 \times |\text{training classes}|}{|\text{testing classes}| + |\text{target classes}|}} \tag{5.2}$$

As coined by [131], where:

Training classes = The number of classes in the training set (including validation set)
Testing classes = The number of classes that are overall tested as well as used during training
Target classes = The number of classes that are used to train on and thus used as the classes that need to be classified

6 DETERMINING THE AMOUNT OF EVALUATION IMAGES

In this chapter, we answer the following sub-question:

How many images are needed, for a statistically reliable evaluation of a multi-class classification?

The ML models in this experiment are chosen based on performance, ease of implementation and inference speed. Furthermore, they are also able to be converted to a Tensorflow Lite model and edge Tensor Processing Unit model, so they can run on the cloud and edge hardware. These models are then trained to identify the 10 classes that are present in the real-world dataset, described in Chapter 5. In training the models, the nature of OOD data is considered, by leaving one distribution out as an evaluation set, and training on four distributions whilst validating on the last distribution set. Due to time constraints, the models that were gathered are pre-trained models. The pre-trained models were refined on the captured dataset through transfer learning.

The number of evaluation images varied between 1 and 80. 10 reruns were done to ensure proper evaluation and to achieve an average accuracy and Confidence Interval (CI) for the certainty of the number of evaluation images found.

6.1 METHODOLOGY

This experiment is structured into two main steps. The first is selecting the Machine Learning (ML) models and training the models as discussed in Section 6.1.2. The second step is the evaluation of the trained ML models in two ways of which the flow can be seen in Figure 13.

First, the ML models were evaluated in more classical ways of evaluation, through a classification report for precision and F1-scores, and a confidence matrix. This was done with all possible evaluation images of the evaluation set. The models were chosen from other research which implements and test the accuracy of these models on similar datasets. For these specific ML models, different input resolutions for the images were required. All of these input resolutions are square, meaning that besides a suitable downsampling technique, a suitable resizing technique was chosen to resize the dataset images from a 4:3 aspect ratio to a square.

Secondly, the variation in the number of evaluation images per class was examined. This was done with multiple reruns per number of evaluation images, which allows for mean value and Confidence Interval (CI)s to be plotted. The CI will be further discussed in Section 6.1.4.

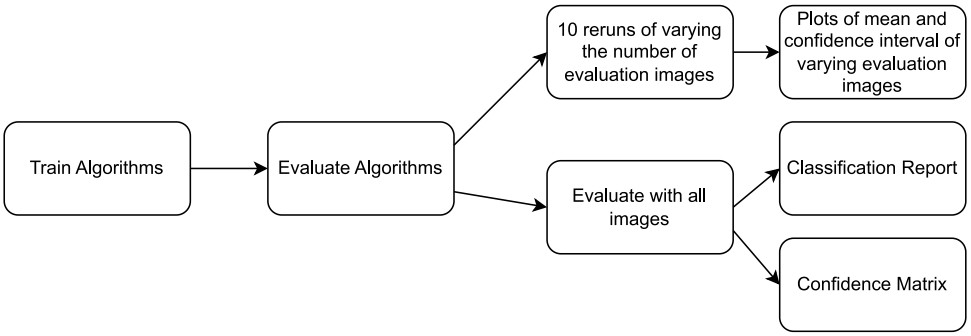


Figure 13: Overview of this experiment and the steps taken.

6.1.1 *Model Choice*

The best-suited Machine Learning models were chosen with the help of a Design Space Exploration (DSE). This DSE is found in Table 12. The criteria that were used, were the classification accuracy based on ImageNet, how easy the ML models are to implement based on their availability in libraries, and time

taken per inference. Better accuracy, easier implementation and faster inference led to ++, while the opposite resulted in - -. The exact overview is given in Table 11.

Table 11: This table explains how the metrics of the DSE are related to the ++/- - awarded in the DSE.

DSE Metric	++	+	+/-	-	- -
Performance	>80%	75-80%	70-75%	60-70%	<60%
Ease of Implementation	Model in libraries	Pre-made model by Google	Libary implementation	Non-libary implementation found	No implementation found
Throughput	<16.66ms	16.66-33.33ms	33.33-60ms	60-100ms	>100ms

The full scores and DSE is given in Table 12. Some of these criteria are more critical for this research, which is why weights were used to indicate the importance of these criteria. ML models were selected as suitable if that ML model scored higher or equal to 9. This number was chosen to allow for a suitable range of ML models to explore, and also to look at secondary criteria, such as the different input ratios and possible quantisation methods to quantize these ML models.

The models which were chosen were MobileNetV2 [132], EfficientNetB0 [133], EfficientNetV2B0 [134], EfficientNetV2S [134] and InceptionResNetV2 [135]. These models were all designed as lightweight ML models, making them more suitable for resource-constrained devices. The layers of these models all exist out of layers that are built of operations which were shown to be able to work and perform well on the chosen hardware [136, 137].

Due to limited time, pre-trained ML models were selected for the analysis. The best would be if these ML models were pre-trained on COCO as it is the best comparable dataset to the one made in this research. The ML models that were found which were pre-trained on COCO were all object detection models, but in this research, the focus is on classification models. Suitable pre-trained classification models where COCO is used as the dataset, could not be found, and thus ML models were used, which were pre-trained on ImageNet.

6.1.2 *Model Training*

Utilising the pre-trained ML models, the models were then retrained using transfer learning on our dataset. For this, the classification head of the ML model was removed, and a new classification head was added. The original weights of the pre-trained ML model were frozen, and only the weights in the classification head were updated.

The ML models were trained with leave-one-distribution-out cross-validation, as illustrated in Figure 14. First of all, one distribution was set aside for the final evaluation, thus five distributions were left to train with. For every evaluation distribution, five separate ML models were trained for every type of model, in which one of the remaining distributions was used as a validation set, and the other four remaining distributions as the training set. To illustrate: for MobileNetV2, where the park distribution was left out, five separate models were trained where all distributions except the park distribution were used as the validation set. These five models were averaged, and this averaged model was used for the final, park distribution, evaluation set. This meant that for every ML model, 6 averaged models were trained, one for every distribution.

The models were trained separately, and the weights of the models were averaged, to better mimic the final deployment. The distribution in which it is placed is not known, and therefore the validation set should be kept separately. This approach mimics federated averaging, which has been proven successful for edge devices placed in different environments [45].

6.1.3 *Model Inference*

After the training of the ML models has succeeded, the test set was used for inference. This refers to the distribution which has not yet been seen by the ML model, ensuring that it is OOD data. This is done ten times for every number of evaluation images. So one image of every class is evaluated, then five and so on. This is done for 1, 5, 10, 15, 25, 50, 75, and 100 images per class.

Table 12: Design Space Exploration of Algorithm Choice (Edge), from "- -" to "++", which represents a score from -2 to +2. This score is multiplied with the weight and finally summed to a total which is shown in the last column.

Algorithm	Performance	Ease of Implementation	Throughput	Total
Weights	**3**	**2**	**1**	
Image Recognition Algorithms for EI and TinyML				
EfficientNet V2 S [134]	++	++	+	11
EfficientNet B0 [133]	+	++	++	9
EfficientNet V2 B0 [134]	+	++	++	9
Inception-ResNet V2 [135]	++	++	-	9
MobileNet v2 [132]	+	++	++	9
Inception v4 [135]	++	+	- -	6
DenseNet [138]	+	+	+	6
MobileNet v1 [139]	+/-	++	++	6
ResNet-50 V1 [140]	+	+/-	+/-	3
ResNet-152 V2 [141]	+	+/-	+/-	3
EfficientNet-EdgeTpu-L [142]	++	- -	+	3
Inception v1 [143]	-	++	++	3
MobileNet v3Small [144]	-	+	++	1
EfficientNet-EdgeTpu-S [142]	+	- -	++	1
EfficientNet-EdgeTpu-M [142]	+	- -	++	1
SqueezeNet [35]	- -	+/-	++	-4
MCUNetV2-H7 [145]	+/-	- -	- - (Not Found)	-6
AttendNets [146]	+/-	- -	- - (Not Found)	-6
RNNPool [147]	+/-	- -	- - (Not Found)	-6

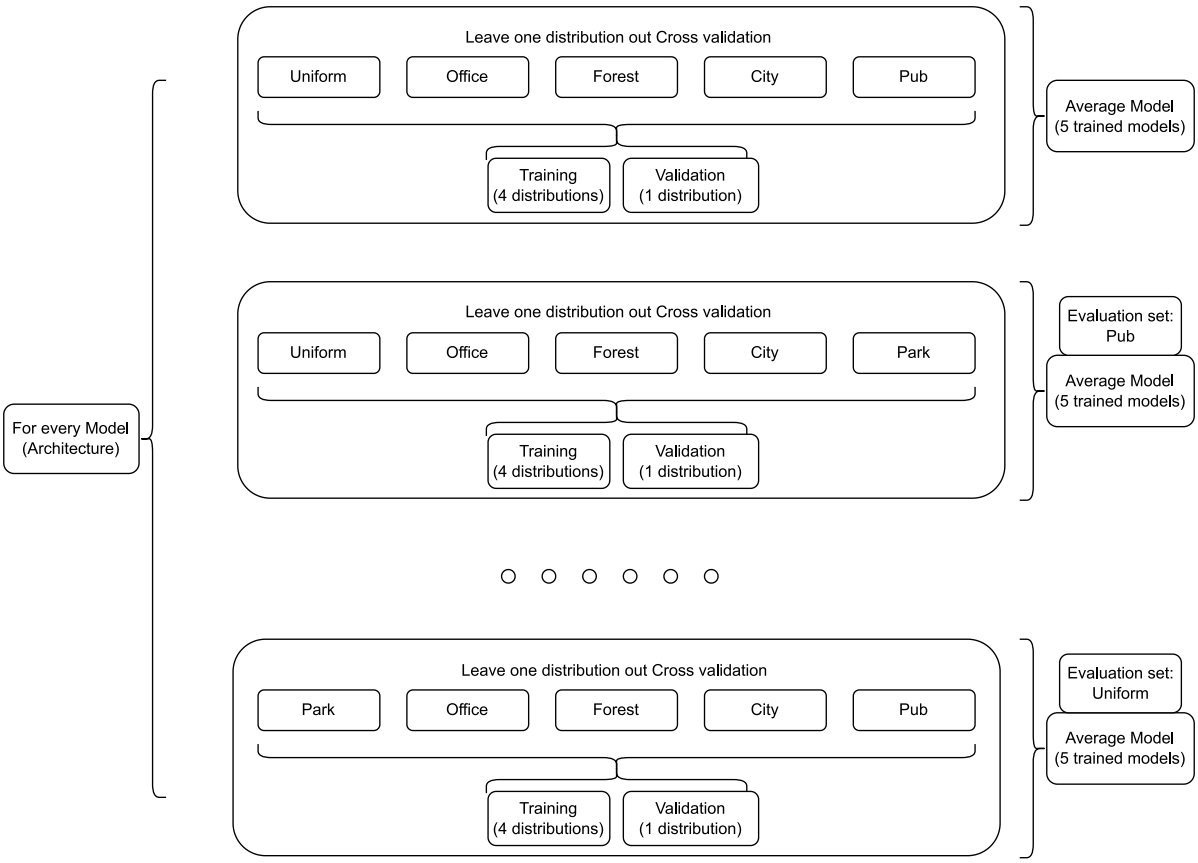


Figure 14: Overview of how training has been done for making averaged ML models which were evaluated for the number of evaluation images.

6.1.4 *Model Evaluation*

The metrics used, are explained in Equations 6.1, 6.2, 6.3, and 6.4. The acronyms that are used, are as follows:

- True positive (TP): when the classifier correctly classifies a Teddy as a Teddy
- False positive (FP): when the classifier classifies a Ball as a Teddy.
- True negative (TN): when the classifier correctly classifies a Ball as a Ball
- False negative (FN): when the classifier classifies a Teddy as a Ball.

The different resulting parameters which can say something about the performance of the classifier are as follows:

$$Accuracy = \frac{TP + TN}{TP + TN + FP + FN} \tag{6.1}$$

$$Precision = \frac{TP}{TP + FP} \tag{6.2}$$

$$Recall = \frac{TP}{TP + FN} \tag{6.3}$$

$$F1 - Score = 2 \times \frac{Precision \times Recall}{Precision + Recall} \tag{6.4}$$

The performance of the classifiers was evaluated using four metrics: accuracy, precision, recall and F1-score. A downside of the accuracy is that if the dataset is unbalanced, it may not give an accurate representation. On the other hand, if the dataset is balanced, it is used as a reliable measure [148].

Next to the more classical ways of evaluating the classifier's performance, for this research, insights were found in the difference between evaluation performances for the different number of evaluation images. An overview of the evaluation is also presented in Figure 15. For this evaluation, accuracy is used as the performance metric. From this performance metric, a mean is taken over ten reruns to finally get a mean and Confidence Interval per number of evaluation images. See Equation 6.5 for how the CI is calculated. This CI gives a confidence measure, that if this experiment is to be repeated, we could be 95% certain that the average accuracy would be within the CI. The CI is calculated to give insight into the variation possible after the evaluation of the varying number of evaluation images.

$$CI = \bar{x} \pm z \times \frac{s}{\sqrt{n}} \tag{6.5}$$

where:

CI = Confidence Interval
$\bar{x}$ = Mean of classification accuracies for the reruns of a specific amount of evaluation images per class.
z = Confidence level value, for a confidence level of 95%, a value of 1.96 is used [149].
s = Standard deviation of the classification accuracies for the reruns of a specific amount of evaluation images per class.
n = Number of reruns of a specific amount of evaluation images per class.

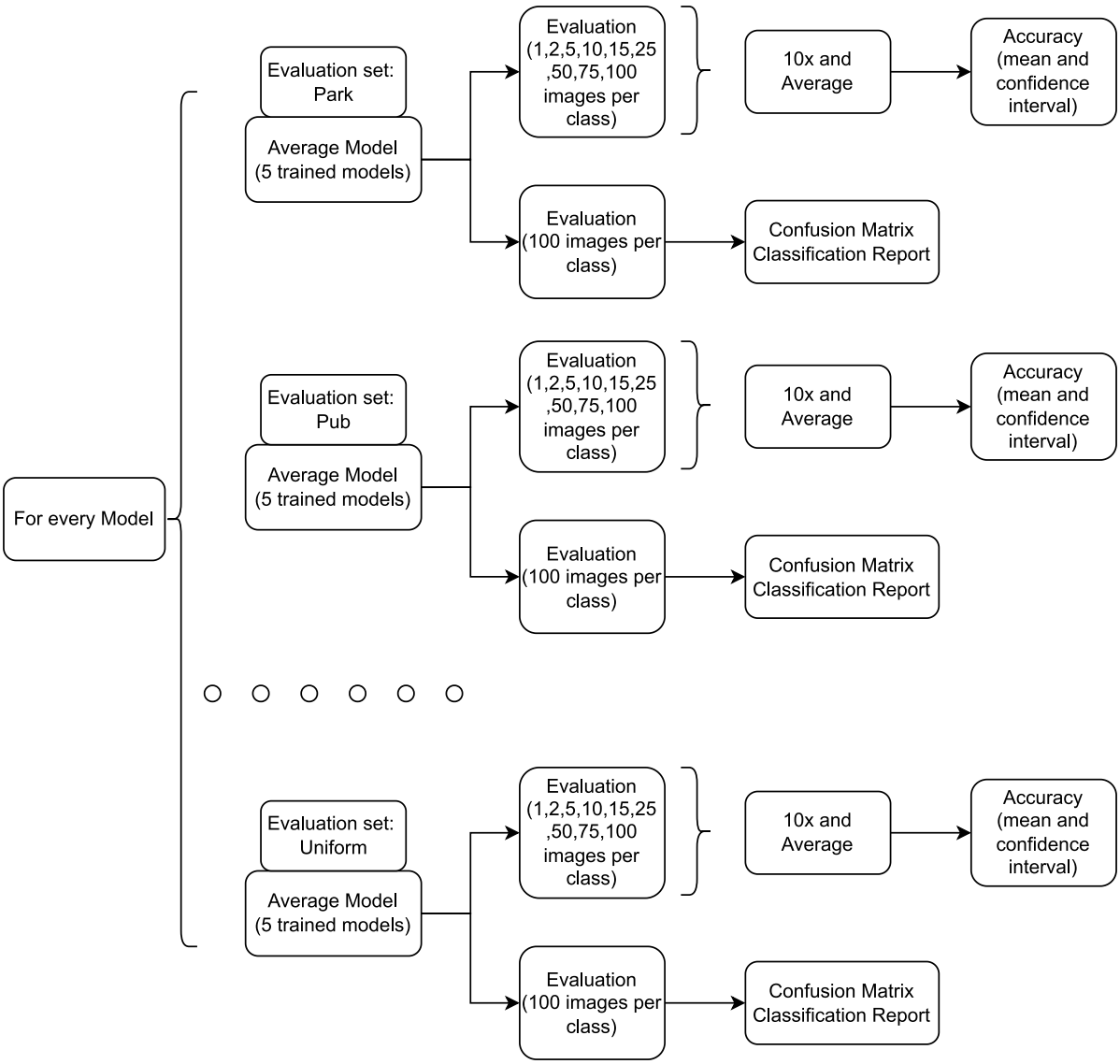


Figure 15: Overview of how training has been done for making models which are reviewed for the number of evaluation images.

6.1.5 *Platform Choice*

In this thesis, custom Python scripts and JupyterLab [150] were used, to train, infer and evaluate the first ML models. To collect results of different runs and evaluation reports, Weights and Biases [151] was used. Weights and Biases allowed for easy access and overview of the trained ML models and their performance by using an online dashboard.

The constructed TF lite models were visualised with the help of Netron [152]. This was done to allow for inspection, to see if the converted TF lite files and edge Tensor Processing Unit-supported TF lite models contained the correct layers. This was done as some layers were not supported by either the TF lite for microcontrollers library or by the edge Tensor Processing Unit (TPU).

6.1.6 *Downsampling Techniques*

The dataset images were pre-processed to the different resolutions needed for the chosen algorithms, which were 224 x 224 pixels for MobileNetV2, EfficientNetB0 and EfficientNetV2B0. For InceptionResNetV2 and EfficientNetV2S, larger images were needed, respectively 299 x 299 pixels and 384 x 384 pixels. Similar pre-processing methods were used for every ML model to ensure that proper comparison

could be achieved. The differences in pre-processing methods are the resizing of the images, as well as the scaling of the RGB values from INT8 to FLOAT32. The dataset was resized with the help of the OpenCV library [153]. Multiple resizing methods were tried, with the nearest neighbour interpolation method being the fastest, while not visually altering the image. The nearest neighbour interpolation method was used to rescale and resize the images at the same time to the desired input sizes of the used ML models, which are 224 x 224 pixels, 299 x 299 pixels and 384 x 384 pixels.

6.2 RESULTS

This experiment resulted in an overview of graphs showing the mean accuracies of the ML models per model and the Confidence Intervals of these mean accuracies if one would repeat the same experiment. The graphs show that the mean accuracy does not differ much for a given number of evaluation images compared to using the entire evaluation set. The graph for EfficientNetV2B0 is shown in Figure 17. The graphs for the other models are given in Appendix A.2.

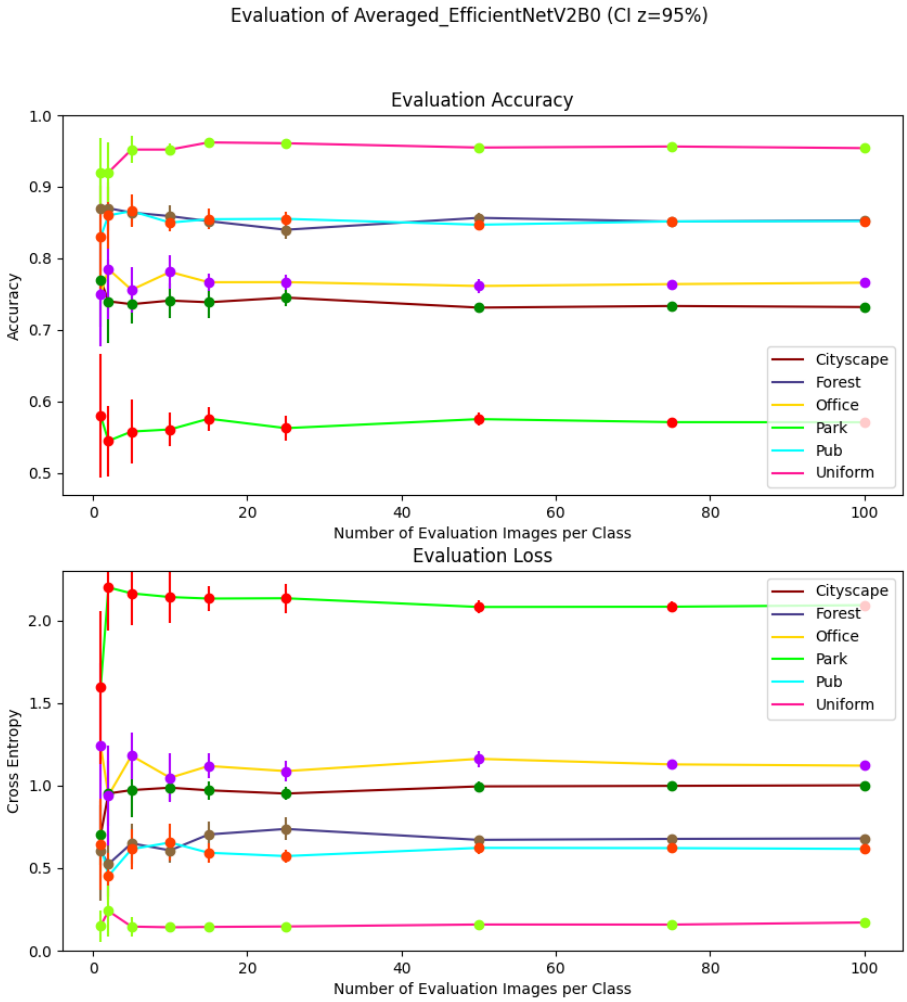


Figure 16: Average accuracy and loss with Confidence Intervals of EfficientNetV2B0.

In Figure 16, the accuracy for the entire evaluation set of a distribution is given. The deviation shown is the accuracy between the different distributions. Here it can be observed that for EfficientNetV2B0, the park distribution is seen as outliers, due to the relative high performance on the other distributions when compared to the park distribution. EfficientNetV2S performs the best of the five models, and MobileNetV2 performs the worst, which is in line with their relative performance on the ImageNet

dataset [132, 133, 134, 135]. EfficientNetV2S performs 6% to 25% better than MobileNetV2 across the different distributions, thus showing that EfficientNetV2S is better in classifying OOD data. Furthermore, InceptionResNetV2 performs worse than the EfficientNet models on more difficult distributions, such as park and office, while achieving better performance in the easier uniform distribution. Performance on the different distributions seems to be in line with what was found in state-of-the-art research [57].

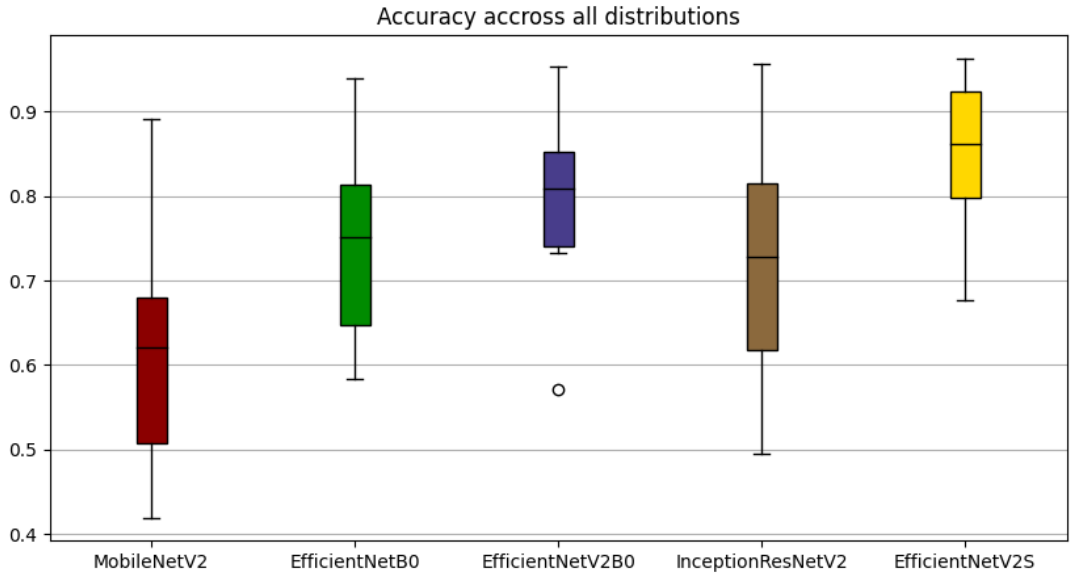


Figure 17: Accuracy across the different distributions for all ML models. The whiskers show the deviation in accuracy across the distributions.

Next to the graph, an overview of all values is given in Table 20 in Appendix A.1 per model and per distribution. The values were calculated by averaging the 10 reruns and retrieving the average classification accuracy. Next to that, with the accuracy of these 10 reruns, the Confidence Interval was calculated. During training, it was observed that the callback of Weights and Biases can take a very long time. This varied a lot but increased the duration of training significantly.

Table 20 and Figure 17 show that the Confidence Interval (CI) decreases when more images are used. However, a reasonable trade-off is situated around the 15 to 25 images mark. Here the CI varies from 1% to 2%. Thus, when this experiment was to be repeated, we are 95% confident that the measured evaluation accuracies vary 1% to 2% from the actual accuracy.

The classification performance was the best for the distribution with uniform backgrounds and the worst for the distribution with park backgrounds. The CI was smallest for the uniform distribution and largest for the park distribution. This is logical, as for the less accurate distributions, there would be more variety in how the models performed, than for more accurate distributions.

6.3 DISCUSSION

The different evaluation distributions performed very differently from each other. The Uniform distribution models were trained on the most cluttered backgrounds and tested on objects with a non-cluttered background, thus improving classification performance. For the Park distribution, the opposite is true, as the background contained the most variety in colours and patterns, which would lead to misclassification.

To reflect on the question posed at the beginning of this experiment, around 20 images per class, the Confidence Interval across all distributions and models was 1% to 2%. This implies that 20 images per class give a very reliable evaluation for a deployed EI device.

A limitation that was introduced, is the fact that the models that were chosen, were pre-trained on ImageNet. This could have affected the final results, as the classes in ImageNet were not the same as the ones used in this dataset. The effects should be limited due to the vastness of the ImageNet dataset. A more representable dataset that could be used for pre-training, is the COCO dataset, due to its vastness in

varying environments. This could potentially decrease the data needed for transfer learning, or increase the potential final performance for a given number of training images.

Another limitation is that the dataset used is a balanced dataset. The data gathered in the real world may be better represented with an unbalanced dataset, where the classes are long-tailed, meaning that for a lot of classes, a small amount of data is available. Comparing the difference in performance would need to be done with the F1-score as opposed to the accuracy metric now used, and it would be seen that the presence of long-tailed classes leads to a loss in classification performance. Another consideration is that for every distribution, the dataset is also balanced. However, it is possible that from some distributions, more data can be gathered than from other distributions, leading to long-tailed distributions as well as the classes present within the distributions.

6.4 CONCLUSION

With the made OOD dataset, 20 images per class can be used for a reliable evaluation. When evaluating on 15 to 25 images per class, the Confidence Interval is reduced to 1% to 2% for a confidence measure of 95%. This means it can be said with 95% confidence that when this experiment is repeated, the final accuracies are within a 1% to 2% margin, where this lower margin indicates more confidence in the repeatability of this experiment.

It was shown that the EfficientNet models perform the best across the different distributions, whereas MobileNetV2 performs the worst. The difference between the models on the most accurate distributions is 6% and the least accurate distributions 25%. All models can be run on an edge device, but EfficientNetV2S and InceptionResNetV2 have a larger memory footprint, mainly due to the fact that they have larger images as input when compared to MobileNetV2, EfficientNetB0 and EffcientNetV2B0.

Furthermore, the park distribution is the most difficult, and the uniform distribution is the easiest to classify for all models. The difficulty in classifying the park distribution was due to the variety in background colours, and the clutteredness of the image. This points to the fact that the surroundings of an edge device should be kept as simple as possible.

7

TRADE-OFF BETWEEN REMOTE EVALUATION AND POWER USAGE

In this chapter, the aim is to create an overview of the trade-off between the remote evaluation of a deployed system, human effort, and the power that must be consumed for such a system to be evaluated, thus answering the following sub-question:

What is the trade-off between evaluation reliability, power usage, and human effort?

The remote evaluation of an edge device is needed, as not all edge devices can be easily accessed by researchers, to retrieve the data. As evaluation is still important to check the proper performance of an edge device, images need to be sent to a central part, to be labelled by one or multiple human experts, to retrieve a ground truth label. As the classification performance of edge devices degrades over time, an evaluation set can be sent, to determine the classification performance over a specified time span.

For the trade-off between evaluation reliability and power usage, a hardware setup was used, to be able to take measurements. The different hardware and software components contributing to the total power usage in the remote evaluation of a deployed device were identified. For this, different cameras were investigated in terms of their power consumption. With the components that are most power-consuming, a model is made that calculates how many images can be sent and how long the device would be able to run.

For the trade-off between evaluation reliability and human effort, a more theoretical approach is taken, where different methods of evaluation are compared. These methods are compared on their accuracy in evaluation and how much effort and time is required from a human expert.

7.1 METHODOLOGY

This section describes the methodology which was followed for creating an overview of all power components and modelling this trade-off. Furthermore, the human effort needed for reliable evaluation is discussed.

7.1.1 *Creating an overview of critical power components*

The camera, inference of images, and transmission of these images are the parts where an edge device consumes power. Firstly, the power consumption of cameras is retrieved from datasheets and can be found in Table 13. Secondly, the power consumption of inference is measured in the next experiment but can be found in Table 17. Thirdly, the power consumption of transmission of images is found in literature, which can be found in Table 16.

The modelling in this experiment consists of making an overview of the impact of different communication protocols and other variables on the prospected battery life of edge devices. The input variables consist of the data size of the image to be sent, how many images are to be sent, and variables of the battery like capacity, nominal voltage, idle power usage and sleep usage. Furthermore, average incoming power may be interesting as often a solar panel is an option that is used for power generation for deployed edge devices.

7.1.2 *Platform Choice*

To write the code needed for power consumption measurements, the Jupyterlab environment from the University of Twente was used for training and inference. This environment hosts multiple GPUs which can be used for machine learning. To be able to compare the computational power of the cloud and the edge, one specific GPU was chosen, which was the NVIDIA Tesla T4 GPU. This specific GPU was chosen as it was found to be the most reliable running the code of this research. The NVIDIA Tesla T4 can achieve around 65 TFLOPS for mixed precision (FP16/FP32) and 130 TOPS for INT8 [154]. This compares to the Google Coral TPU Development Board where the CPU has 2.3 TOPS [155], and the TPU has 4 TOPS [137].

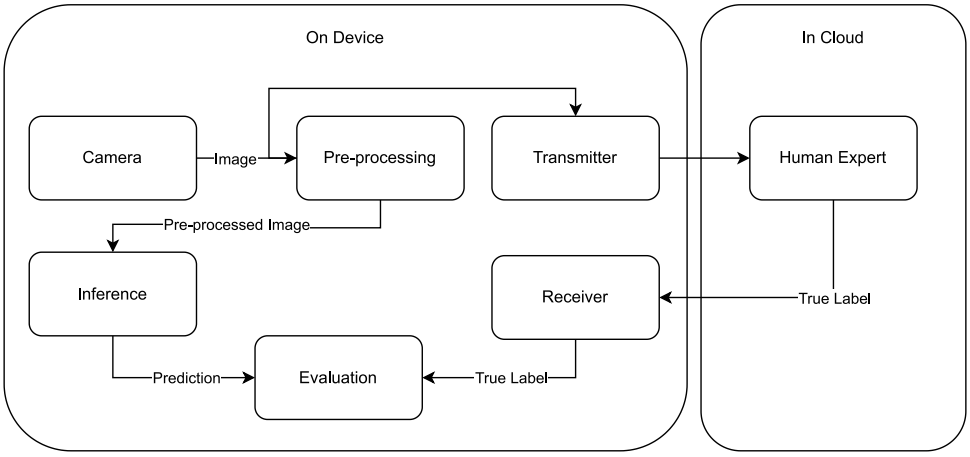


Figure 18: Flow of data in a deployed device. The different steps consume varying amounts of power, with the transmission of data being the most power-consuming step.

Together with Jupyterlab, Windows Subsystem for Linus (WSL) [156] was used to use the edge TPU compiler [157] of Google Coral. This edge TPU compiler is used to compile quantised models to be used on the edge TPU on the Google Coral TPU development board.

7.1.2.1 *Camera Choice*

For the chosen Google Coral TPU Development Board, there are different ways of connecting the camera, which are the MIPI Camera Serial Interface 2 (MIPI CSI-2) connector and the Universal Serial Bus (USB) connector. Table 13 shows there is little difference between the cameras, except for the throughput of the MIPI CSI-2 camera connections. These can be used in later revisions where classification models get more efficient and a higher resolution image can be used as input. The other significant difference is power usage. However, this differs significantly between USB cameras, so depending on the resolution and framerate needs, a less power-consuming USB camera can be chosen.

The documentation for edge TPU benchmarks [137] shows that the largest model expects an input image of 513 by 513. This resolution is nowhere near the restricting limits of both types of cameras. Thus, the bandwidth for both should be more than enough at this moment. Next to that, in the edge TPU benchmarks table, it can also be seen that the inference time for nine models is low enough to run at 60fps. For two other models, which are larger models, 30fps is possible. The larger models do have higher accuracy. Due to the high bandwidth and availability, the Coral AI camera was chosen for this research. This camera is used as an indication of the power consumption of the camera subsystem.

Table 13: Design Space Exploration for Camera Choice.
For the MIPI CSI-2 cameras, the Coral AI Camera [158], and the e-con Systems Coral cameras [159] were used, and as a reference for the USB camera the Logitech C920 [160] was used.

Camera Type	Bandwidth	Max Resolution	FPS for FullHD	Power usage min (W)	Power usage max (W)
MIPI CSI-2 Cameras					
Coral AI camera	320MB/s per lane (4 lanes)	2592 x 1944	30	Not found	Not found
e-CAM30_CUCRL	320MB/s per lane (4 lanes)	2268 x 1512	60	0.79	1.22
e-CAM50_CUCRL	320MB/s per lane (4 lanes)	2592 x 1944	65	0.840	1.160
USB Cameras					
Logitech HD Webcam C270	40MB/s	1600 x 1200	30	Not found	2.5W
Logitech QuickCam pro 9000	40MB/s	1280 x 720	30	Not found	Not found
Microsoft LiveCam Studio	40MB/s	1920 X 1080	30	Not found	Not found

7.1.2.2 *Quantisation Techniques*

To be able to run models on the edge TPU, these models need to be fully integer quantised [161]. This allows the models to fit on the hardware, as well as be compiled by the edge TPU compiler. However, this may also influence the final accuracy of the model. This variation of accuracies is discussed and further investigated in the experiments in Chapter 8.

7.1.2.3 *Power Consumption Evaluation*

The separate power-consuming parts of the data flow need to be characterized, which may be hard due to missing data. To get the power requirements, a current measuring device is used to get the idle current consumption of the device. A separate voltage measurement is taken to determine the power consumption, which can ultimately be compared. It was not possible to measure different communication protocol modules, thus theoretical values were taken from other research, which is shown in Table 14.

Table 14: Comparison between different long-range communication techniques, which are a possible use case for edge devices. Values were obtained from research and datasheets which are given in the references column.

Communication Type	Uplink Speed	Downlink Speed	Power (J / kbyte)	Range	Latency	References
LoRaWAN	250 bit/s - 11 kbit/s	250 bit/s - 11 kbit/s	0.698	10 km	250 ms	[162, 163]
4G	100 Mbit/s	100 Mbit/s	0.217	25 km	25 ms	[164, 165, 166]
LTE-M	7 Mbit/s	4 Mbit/s	0.27	10-50 km	10 - 15 ms	[167, 168]
NB-IoT	159 kbit/s	127 kbit/s	0.198	1-10 km	1 - 10 s	[167, 168]
Sigfox	600 bit/s	600 bit/s	5.906	10-50 km	20 s	[169, 170, 171]
Wi-Fi	80 Mbit/s	25 Mbit/s	0.385	50 m	1-3 ms	[172, 173]

These values can be used to calculate the amount of Joules needed to send one 36.0KB image and how many seconds it takes to send a 36.0KB image. A 36.0KB image was assumed, as this is the approximate size of a 224 X 224 pixels image. To make these calculations, Equations 7.1 and 7.2 were used.

$$P_I = P_{KB} \times S \tag{7.1}$$

where:

P_I = Power per Image (J)
P_KB = Power per KByte (J / KB)
S = Image size (KB)

$$T_T = \frac{S}{T_S} \tag{7.2}$$

where:

T_T = Time per Image (s)
T_S = KByte per seconds (KB / s)
S = Image size (KB)

7.2 RESULTS

There are two main results that are gathered from this experiment. The first part consists of the trade-off between evaluation and power consumption. A graph was rendered to show the different communication protocols and their life span on a given battery capacity. The second part compares the human effort and the evaluation reliability, where a theoretical approach has been taken.

7.2.1 *Creating an overview of critical power components*

As shown in Table 15, the largest part of the power consumed is the sending of images. In the table, the minimum and maximum power consumption for every aspect is shown. As can be seen from the table, the largest part of the power consumed is the transmission of images for feedback.

Table 15: The power consumption and relative power consumption of the different parts of a EI device. Power consumption is calculated for 1 image. For camera power consumption, the power consumption in Watt and the frames per second from Table 13 can be used to calculate the amount of Joules used per image. For pre-processing and inference, the values from Table 17 are used. For feedback transmission, the values from Table 16 are used.

System Part	Power consumption (J)	Power consumption (% of total)
Camera	0.013 - 0.020	0.007% - 0.000%
Pre-processing and Inference	0.031 - 12.423	0.016% - 0.210%
Feedback Transmission	198 - 5906.25	99.978% - 99.790%

The power requirements for each tested communication technology are shown in Table 14 and visualized in Figure 19. Calculations are done to give an overview of how much power is required to send one picture. The power requirements per image are presented in Table 16.

Table 16: Comparison between power requirements for a 36.0 KB picture to be sent by different communication protocols.

Communication Type	Power (J / kbit)	Power (J/image)	Tx time (s)
LoRaWAN	0.087	25.13	26.18
4G	0.027	7.81	0.35
LTE-M	0.0338	9.72	0.04
NB-IoT	0.0248	7.13	1.81
Sigfox	0.738	212.63	180
Wi-Fi	0.048	13.86	0.0036

In Figure 19, Sigfox stands out, as it seems to have particularly high power consumption. This is due to the fact that Sigfox, as a communication protocol, is used for small amounts of data and is not made for sending larger data like images. This is why sending an image would take so long, and there is such a high-power draw. LoRaWAN has been made for sending small amounts of data over large distances. This is why the transmission time is so high. However, it compensates for this with a low J / kbit rating.

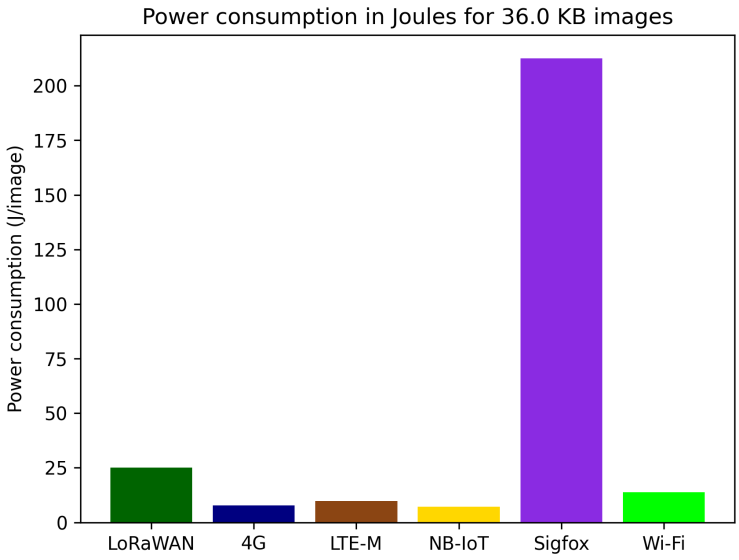


Figure 19: A bar chart displaying the different power consumption for transmitting a 36.0 KB image using different protocols.

7.2.2 *Modelling Battery Capacity vs Photos sent and Time*

A model was made based on the different transmission protocols, as well as different inputs, like battery capacity, uptime, power usage of the different software parts, and the number of photos sent per day. This model shows how different protocols lead to different maximum lifetimes per a given battery size. In Chapter 10, this model is used to create an overview of the battery charge for a typically deployed edge device. A battery capacity of 72Wh is considered a reasonable size, as this is 3 Li-Ion batteries in series with 2 parallel, which has been used in wild-life camera applications. The nominal and idle power usage is evaluated from the datasheet of the development board and retrieved from measurements. Next to that, an uptime of approximately 0.25% per day is assumed, as is calculated with the transmission time of the images, as well as inference time. With the given image size of 36.0KB and a battery capacity of 72Wh, around 36400 images can be sent by NB-IoT, which is the most efficient protocol. 4G is the second most efficient protocol, which is able to send around 33200 images. The worst transmission protocol is Sigfox, which can only send around 1220 images.

7.2.3 *Human Effort*

When the images are sent back for evaluation, more images often relate to higher evaluation reliability. Thus human experts would need to check more images, thus increasing human effort. Multiple human experts labelling the same evaluation set, leads to a higher reliability in evaluation, as opposed to one human expert [174]. In Chapter 6, it was shown that for the given dataset in this research, only about 20 images per class are needed for reliable evaluation, thus decreasing the human effort. A downside of this is the fact that this is for just one deployed EI device. Often multiple edge devices are deployed at once, thus increasing the human effort again.

This can be counteracted by using a teacher-student model [13, 175]. A large cloud model is deployed as a teacher model together with the smaller edge devices as students. The cloud model can then infer the images and predict which label would be associated with the image. This cloud model requires no transmission of images to check its classification performance, and these cloud models are often more accurate than edge models [49, 50]. This system can be used in place of a human expert, or alongside a human expert to check the work of edge devices.

7.3 DISCUSSION

This experiment has shown that communication protocols such as Wi-Fi indeed take a lot of power to send an image, yet, it does so very fast. More typical communication types in IoT consume less power per image sent but take longer. NB-IoT and 4G are suitable protocols, as they have the lowest transmission power while sending the image. Furthermore, they do not take long to send the picture, but 4G is faster than NB-IoT.

A fascinating insight is found in Sigfox, as it is assumed widely to be energy efficient, which did not appear to be the case. This is mainly because sending a small amount of data takes a long time, as an image is too large for this technology. Comparing the amount of power needed to send images, the rest of the components in the device have insignificant power usage, as shown in Table 15.

For a battery of 72Wh with no incoming power, with the most power-efficient communication protocol, NB-IoT, around 34600 images can be sent. If we retrieve the number of images needed for reliable evaluation from the previous chapter, 20 images are required per class. In the case of the 10 classes in this research, this equals 200 images. Comparing this to 34600 images that could be sent, 173 evaluation sets could potentially be sent. This means that during deployment, besides inference, an evaluation set can be sent back to a central location to evaluate the current classification performance of a deployed edge device. With the large amounts of power, updating the machine learning model by way of fine-tuning can be considered to increase the classification performance over time.

To minimize human effort, while keeping reliable evaluation, the number of evaluation images per edge device can be reduced to 20 images per class. Using the combined knowledge of multiple human experts can lead to a better evaluation, thus increasing evaluation reliability. Here the trade-off lies with the fact, that this would increase the human effort again.

A limitation of this experiment is that all power consumption values of the transmission protocols are theoretical values and are not the power consumption of a radio or radio module when deployed in the real world. This could lead to the values provided in this experiment being too low, as there may be more power drawing components in a radio module than just the radio, or too high, as the values given are theoretical maximum values. Measuring the power draw would lead to more accurate measurements when an edge device is deployed, ultimately leading to better insight into possible trade-offs between the cloud and the edge.

7.4 CONCLUSION

The optimal communication protocol to use seems to be NB-IoT, with a power consumption of 7.13 Joule per 36.0 KB image and a relatively fast transmission time of 1.81 seconds per image. 4G has a slightly higher power consumption with 7.81 Joule per image but could be more suitable due to a decreased transmission time of 0.35 seconds.

In the best scenario, around 36400 36.0 KB images can be sent with a battery of 72 Wh with NB-IoT. Experiment 1 in Chapter 6 showed that around 20 images per class are needed for reliable evaluation. This signifies that reliable evaluation can be attained with NB-IoT. For the given settings for NB-IoT, every transmission protocol, except for Sigfox, can send enough images for reliable evaluation, as Sigfox can send 1220 images on a 72 Wh battery.

Concerning human effort, 20 images per class need to be labelled by human experts. This has to be done for every edge device, increasing the human effort again. However, the human effort can be reduced by employing a student-teacher model structure by having a larger, more accurate cloud model label the evaluation images and computing accuracy. The trade-off which is introduced here is that this could lead to inaccuracies, as the teacher model could falsely predict the evaluation images from the edge, whereas a human expert is less likely to label the images wrong.

8

TRADE-OFF BETWEEN INFERENCE ON EDGE CPU OR EDGE TPU AND QUANTIZATION

On an edge device, besides a CPU, a machine learning accelerator like a TPU can be used to infer images. The architectural differences between these components can lead to differences in classification performance. Besides the architectural differences, the models which run on these components are different, which can also influence performance. To investigate the influence of the different architectures on classification performance, the following question is answered in this chapter:

> *What is the trade-off between inference on edge CPU or edge TPU, and how does quantisation influence this?*

To explore the trade-off between these architectures, the averaged ML models that were trained in Chapter 6 are quantised into TF lite models. These two models were then run on the cloud hardware. Next to that, the quantised model was compiled to be run on the TPU. The quantised model was run on the CPU, and the TPU compiled models were run on the TPU of the model. This classification performance difference is interesting to investigate, as quantised models can sometimes perform better due to better generalisation [14].

The ML models were all evaluated on classification performance with the full evaluation set for every distribution and for inference time. The power consumption of the edge device was also measured when running inference on the CPU and TPU. With the classification performance and the power consumption, a trade-off can be made, which ML model and which hardware can best be used.

8.1 METHODOLOGY

In this section, the different steps are explained, which are taken to explore the trade-off between classification performance and power consumption for different hardware architectures, and their respective ML models. The models to be investigated are MobileNetV2, EfficientNetB0, EfficientNetV2B0, EfficientNetV2S and InceptionResNetV2.

The models are run on the cloud GPU, edge device CPU and edge device TPU. To be able to run the ML models on the edge device CPU, they need to be quantised and to run the ML models on the edge device TPU, the quantised models need to be compiled for the TPU.

However, the current version of the edge compiler, 16.0.384591198, could only be used for MobileNetV2, EfficientnetV2B0 and InceptionResNetV2. An older version, version 15.0.340273435, could be used for EfficientNetB0. However, for EfficientNetV2S, no edge TPU compiler version worked, and no errors were mentioned. This is why for the parts of this experiment concerning running the model on the edge TPU, the EfficientNetV2S model is missing.

These ML models are evaluated on classification performance and inference time. The power consumption for the edge device is also measured for the models which are run on the edge device.

8.1.1 *Model Inference and Evaluation*

The JupyterLab [150] environment of the University of Twente was used to generate the different models and run inference and evaluate them. The Google Coral TPU development board was used for the edge compiled devices, together with a large SD card to read the images from. This edge device was chosen after a Design Space Exploration (DSE), as explained in Section 4.2.

8.1.2 *Experiment*

The ML models that are used, are the same models which were trained in Chapter 6 and are thus trained in the same way as explained in Section 6.1.2.

When running the inference of the models, the following metrics may be measured during the tests:

- Accuracy
- F1-score
- Power Consumption (W)
- Inference speed (s)

The evaluation set that was used for the distributions consisted of all available images, thus 100 images for 10 classes per distribution. This is the same as mentioned in Section 6.1.4, with the exception that the number of evaluation images was not varied. To attain one value for the different ML architectures, the results of all models are summed in one confusion matrix. For example, all six confusion matrices for MobileNetV2 are summed, to calculate the classification performance metrics.

The current measurements are done with a Keysight 34461a Digital Multimeter (DMM) [176]. This DMM can log the current values during inference. The current was measured at the power supply of the development board, thus measuring the total power consumption of the edge device. The current measurements are taken 10 times for every ML model that runs on the edge device. These measurements are then used to calculate an average current measurement for every model.

These measurements were taken over a few seconds time when the models were inferring images, and 100 measurements were taken in this period which was averaged by the DMM. Together with the earlier measured voltage, the total power draw is calculated. The time it takes to infer an image is also essential, as this may make a difference in power requirements. Inference time was timed in the code itself. The power consumed per image is calculated as shown in Equation 8.1.

$$P_I = P_W \times \frac{T}{1000} \tag{8.1}$$

where:

P_I = Power per Image (J/image)
P_W = Power (W)
T = Time taken for inference (ms/image)

8.2 RESULTS

The final results of this experiment are the classification performance results for the different cloud and edge hardware and quantisation levels as shown in Figure 20. More detailed results can be found in Appendix B.1 in Tables 21 and 22. These performance metrics were retrieved from the summed confusion matrixes as seen in Appendix B.2.
As shown in Figure 20 and Table 21 and 22, EfficientNetV2S was the best-performing model. It outperformed the other complete models in the cloud with 6% to 22%, the other quantised models in the cloud 8% to 25% and the other quantised models on the edge CPU with 9% to 24%. However, EfficientNetV2S could not be converted to be used by the edge TPU and was omitted for the following experiments. This indicates that for the classification on the edge TPU, EfficientnetV2B0 performed the best, outperforming the other models with 8% to 24%. After EfficientNetV2S, EfficientnetV2B0 was the model which performed the best across all devices and levels of quantisation. The outliers which are present for the EfficientnetV2B0 models evaluated on the cloud are both evaluations where the park distribution was the holdout set.
As is seen in Table 17, the power requirements per model per device and quantisation level are calculated, showing that inference time was the biggest influence on power requirements. MobileNetV2 was the cheapest regarding both time and power consumption per image, with an inference time of 133 ms and 0.5 Joule per image for inference on the edge CPU, and 8.5 ms and 0.03 Joule per image on the edge TPU. Inference times were high for both EfficientNetV2S and InceiptionResNetV2, which took at least ten times longer than other models, respectively 2570 ms and 10.5 Joules per image, and 3275 ms and 12.5 Joules per image. EfficientNetB0 and EfficientNetV2B0 had a 2 to 3 times higher time and power consumption per image as compared to MobileNetV2.

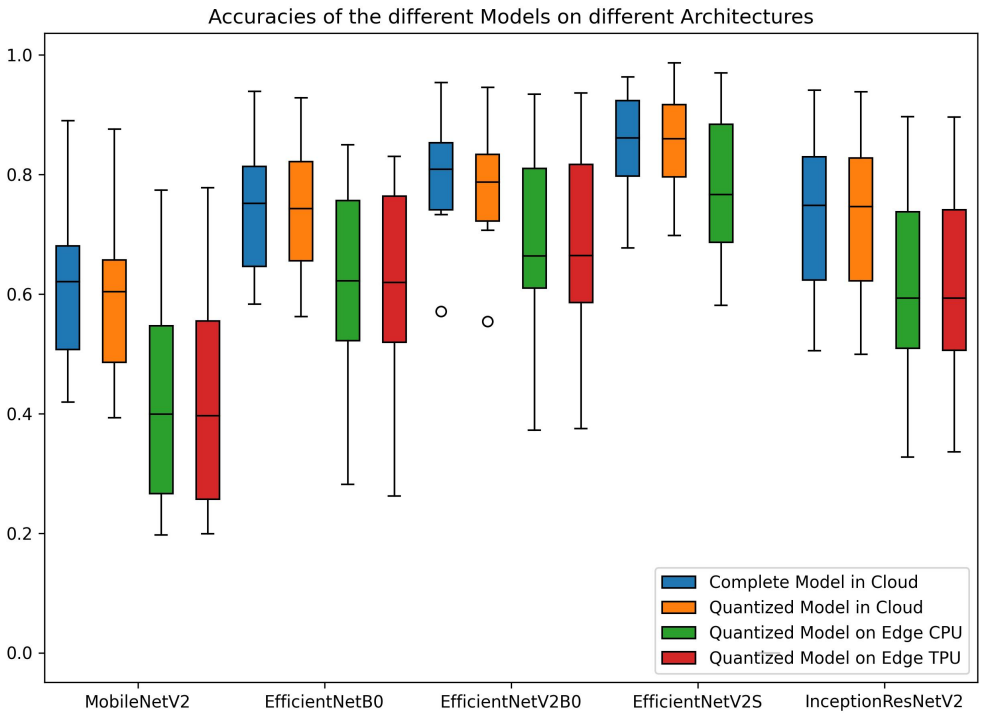


Figure 20: The results for the different models on the different architectures. The whiskers denote the variation in accuracy for the different distributions in the holdout set. The results are shown for the model and quantised model in the cloud, as well as the quantised model on edge CPU and the compiled quantised model on edge TPU.

Table 17: Power consumption and inference time of models on the edge device. Here the quantised models are shown running on CPU and TPU. A metric for power consumed per 36.0KB image inferred is given.

Algorithm	Quantised running on CPU			Quantised running on TPU		
	Power (W)	Time (ms/image)	Power (J/image)	Power (W)	Time (ms/image)	Power (J/image)
MobileNetV2	3.726	**133.609**	**0.497**	3.604	**8.563**	**0.031**
EfficientNetB0	4.053	280.544	1.137	3.993	24.126	0.096
EfficientNetV2B0	3.947	270.294	1.067	3.833	19.709	0.076
EfficientNetV2S	4.108	2573.29	10.571	N.A.	N.A.	N.A.
InceptionResNetV2	3.789	3278.6	12.423	3.903	141.246	0.551

8.3 DISCUSSION

As most of the literature has said [49, 50], when quantisation occurs, the accuracy drops for all models. This is expected as one misses the preciseness of numbers when moving from floating point numbers to integers. However, a specific limitation was found when moving from the cloud to the edge, as the same Tensorflow Lite file seems to have different accuracy when deployed, with the accuracy difference ranging from 8% to 15%.

This problem has been investigated, and the same TF lite file seems to have different values when loaded for some but not all neurons. This may occur due to the differences in mapping as in the cloud, a GPU is used, and on the edge, a CPU and TPU. Another explanation would be the differences in the libraries used. The official Tensorflow Keras library was used in the cloud, while on the Google Coral

TPU Development Board, the official Coral Tensorflow library was used. A last reason could be that the models used are optimised to be run on the cloud instead of on the edge.

The consequence of the large drop in performance when moving the models from the cloud to the edge is the fact that in later experiments, the edge performed better than the cloud. The trade-off could then lead to the edge being more suitable than the cloud as it performs better, even when the cloud-trained models perform a lot better on the cloud.

To reflect on the question posed, running the ML models on the cloud leads to consistently higher performance than running the models on the edge. This drop in performance could not be counteracted. When deploying the ML models on the edge, the accuracy difference in running the models on the edge CPU and TPU is trivial, while performing the inference on TPU leads to significantly lower power consumption. Thus when ML models are run on the edge, using a machine learning accelerator such as a TPU is preferred.

8.4 CONCLUSION

When comparing classification performance, EfficientNetV2S was the best-performing model, outperforming the other models across different devices and quantisation levels 6% to 25%. This model could not be compiled for the edge TPU, thus leading EfficientNetV2B0 to be the best-performing model when deployed on edge TPU, outperforming the other models with 8% to 24%.

Finally, when comparing power requirements when the models are deployed on the edge, MobileNetV2 performs best with a power consumption of 0.03 Joules per image on the edge TPU, with EfficientNetB0 and EfficientNetV2B0 following closely, with 0.10 and 0.08 Joules per image respectively. Conversely, EfficientNetV2S and InceptionResNetV2 both perform significantly worse, with InceptionResNetV2 having a power consumption of 0.55 Joules per image on the edge TPU. The difference in power usage per inference primarily resides in the difference in time inference takes, so lowering the inference time could lead to more power-efficient models.

9

DIFFERENCE IN PERFORMANCE BETWEEN RUNNING CLOUD FINE-TUNED MODELS IN THE CLOUD AND ON THE EDGE

In this experiment, we focus on fine-tuning techniques in the cloud and running these fine-tuned models both in the cloud and on the edge [19]. To address the drop in accuracy that can occur when deploying cloud-trained models on the edge, we investigate the following sub-question:

> *What is the trade-off between inference of cloud fine-tuned models on cloud and edge, and how does quantisation influence this?*

We fine-tuned the ML models in three different ways: retraining the last layer from randomized weights, tuning the weights of the last layer with a small learning rate, or tuning the weights of the entire model with a small learning rate. We showed in Chapter 6, that for the dataset presented in this thesis, 20 images per class are needed for a reliable evaluation, so the other 80 images per class can be used for fine-tuning the model. We show the cloud fine-tuned models performance on the cloud GPU and the edge TPU.

9.1 METHODOLOGY

In this section, the methodology is explained, which has been used to compare running cloud fine-tuned models both in the cloud and on the edge. It describes which models are used, in what different ways they are fine-tuned, and how these fine-tuned models are evaluated.

9.1.1 *Model Choice*

We used the MobileNetV2, EfficientNetB0 and EfficientNetV2B0 models. The EfficientNetV2S model cannot be run on the edge TPU, thus this model is not further investigated. Secondly, fine-tuning on the edge is limited to models which have an output embedding consisting of a vector length of 1280, thus to be able to compare results, InceptionResNetV2 was excluded from this experiment.

9.1.2 *Model Fine-tuning*

The models are fine-tuned in three different ways, as these are primarily used. No names for these fine-tuning strategies could be found, so they are named as follows: Last Layer Randomized weights Training (LLRT), Last Layer Small Learning Rate (LLSLR) and All Layers Small Learning Rate (ALSLR). Firstly, fine-tuning by randomising the weights of the last layer and training this last layer again, which is LLRT [177]. Secondly, fine-tuning can be done by changing the learning rate to a small learning rate and keeping on learning with the old weights [178, 179]. Using a small learning rate was done in two ways. LLSLR considers only training the last layer with a small learning rate and freezing the rest of the weights. ALSLR consists of unfreezing the weights and training all layers with a small learning rate.

Fine-tuning these models was performed while varying the number of training images, which can be seen in Figure 21. This is done with training on 1, 5, 10, 15, 20, 30, 50, and 80 images per class. As our dataset had 100 images per class per distribution, a maximum of 80 images per class were used for fine-tuning.

9.1.3 *Model Evaluation*

Evaluation of the models was done with a 5-fold cross-evaluation, as shown in Figure 21. Using a 5-fold evaluation, 20 images were used for evaluation of every fold. These results were summed for every number of training images per distribution. Thus all evaluations for the park distribution, fine-tuned on 1 image per class, were summed, from which a summed confusion matrix could be made. This summed

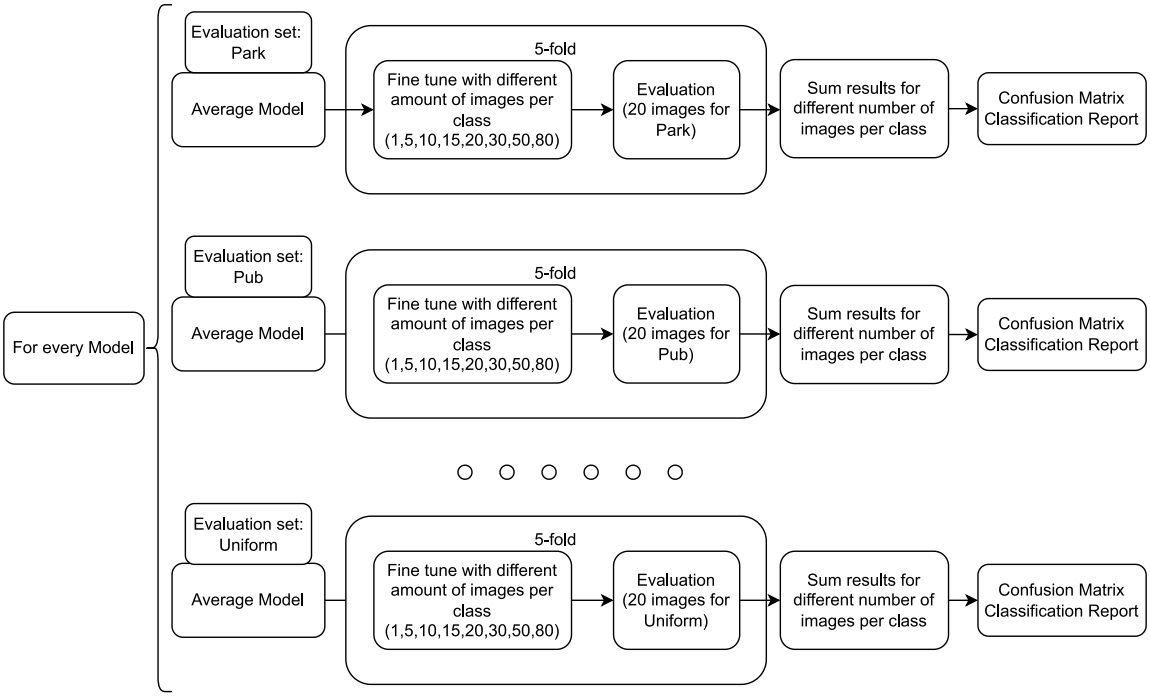


Figure 21: Overview of how training has been done for fine-tuning the ML models that were fine-tuned with varying numbers of training images.

confusion matrix was used to retrieve the classification performance of using a specific amount of training images for fine-tuning.

9.1.4 *Experiment*

The method presented in Figure 21 was followed. The accuracy of the different models and fine-tuning methods was retrieved and compared against each other. To calculate the accuracy, the confusion matrices for every number of fine-tuning images in a distribution were added together. This means that all five folds are combined into one accuracy for every number of fine-tuning images per distribution.

Then for every number of fine-tuning images, the average accuracy, as well as the standard deviation, was calculated across all six distributions. This gives an overview of how well the different models performed after fine-tuning on the different distributions, and how the performance varies between the distributions for a given number of fine-tuning images

9.2 RESULTS

The result of running the fine-tuning in the cloud as compared to on the edge can be found in Appendix C in Tables 23, 24. The results of the fine-tuned models are also illustrated in Figure 22. In this figure, only EfficientNetV2B0 is shown, as it was the best-performing model, as seen in the previous experiment. The figure also shows the standard deviation in performance between the domains for a given number of fine-tuning images per class. The other models are shown in Appendix C in Figure 52.

In Figure 22 and 52, it can be seen that the overall performance of the cloud fine-tuned models on the cloud is significantly better than the cloud fine-tuned models on the edge, with a maximum drop of 25%. It can also be seen that when fine-tuning with the LLRT method, fine-tuning with 1 image per class, greatly improves performance for all models. The other methods for fine-tuning, LLSLR and ALSLR, also experience an increase in performance, but this only occurs after more images are used for fine-tuning.

The classification performance increases by around 6% for every cloud fine-tuned model run on the edge when 80 images per class are used for fine-tuning. The exception is EfficientNetV2B0 when using the ALSLR method. Here, besides the performance drop that was already seen, there also seems to be another drop in accuracy of 12% after 1 to 5 images per class. After fine-tuning for more than 20

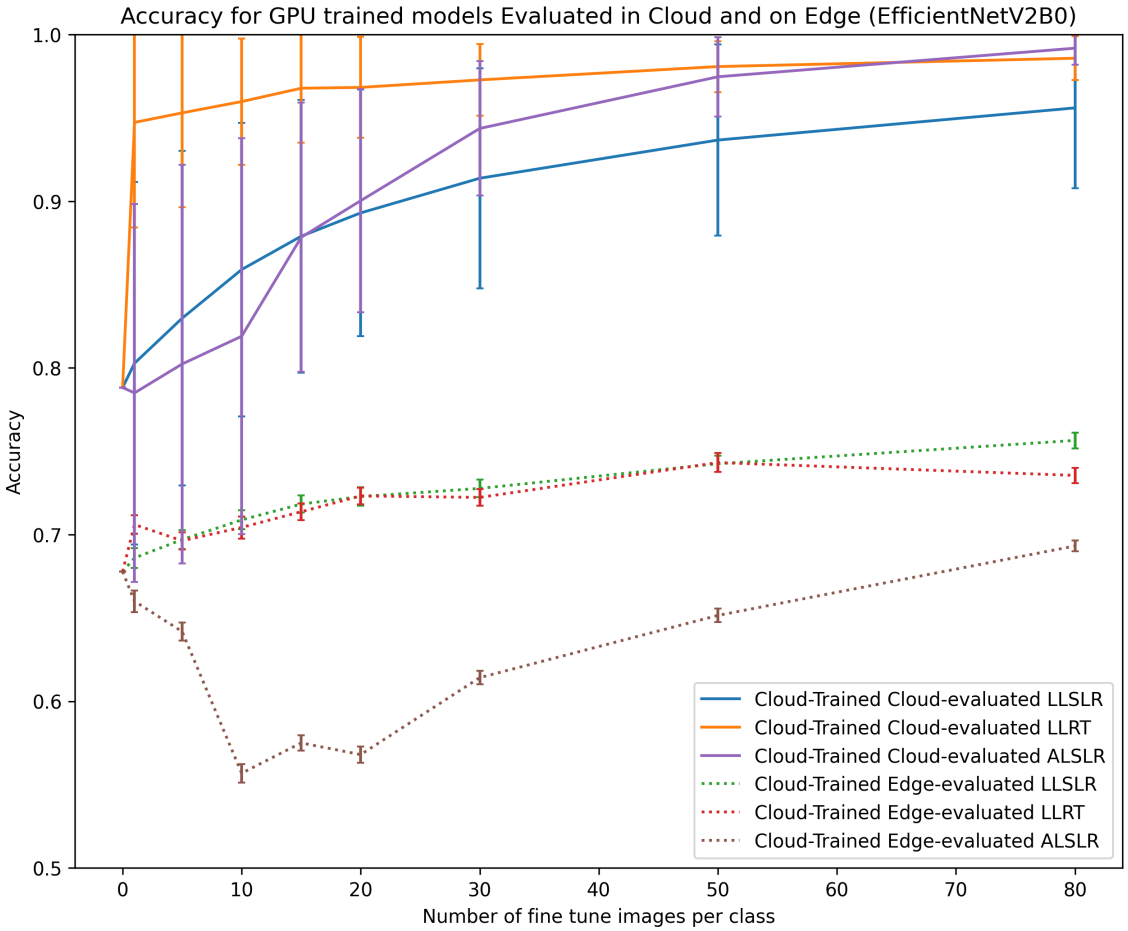


Figure 22: The results for EfficientNetV2B0 fine-tuned models trained on the cloud and tested on both cloud and edge to investigate the impact of testing on different platforms. The accuracy is given for the varying amounts of fine-tuning images. The accuracy of 0 images per class is given by the non-fine-tuned models as found in Chapter 8.

images per class, this performance drop is mitigated but only returns to its original performance when fine-tuning on 80 images per class.

9.3 DISCUSSION

From the results of this experiment, it can be seen that similar to the drop in performance we saw in the previous experiment, a drop in accuracy of 25% can be seen when moving the same models from inference in the cloud to inference on the edge. The drop in accuracy is larger in this experiment when compared to the previous experiment, as the accuracy in the cloud was increased due to fine-tuning. We think this drop is due to the same reasons as in the previous experiment, as the method of moving the models from the cloud to the edge stayed the same.

Next to the drop in accuracy when moving from the cloud to the edge, EfficientNetV2B0 experiences another drop in performance for the ALSLR method when deployed on the edge. This suggests that this model is not as robust against the ALSLR fine-tuning method as compared to MobileNetV2 and EfficientNetB0. However, the model performs well on the cloud, indicating that differences in libraries or architecture may influence performance on the edge.

9.4 CONCLUSION

On the edge, there was an increase of around 6% for the cloud fine-tuned models when fine-tuning for 80 images per class, except for the ALSLR method for EfficientNetV2B0 which had similar accuracy.

Similar to the previous experiment, when fine-tuning the models in the cloud, the model converted to the edge drops 25% in classification accuracy. This drop in performance was greater than the performance drop observed in the previous experiment due to the models in the cloud being more accurate after fine-tuning than the models used in the previous experiment. The drop in performance is thought to be due to the same reasons as the drop in performance seen in the experiment discussed in Chapter 8.

Furthermore, for the EfficientNetV2B0 model, another drop of 12% has been found for the ALSLR fine-tuning method, which is not seen for MobileNetV2 or EfficientNetB0. As the models run on the cloud perform well, this can be attributed to the difference in libraries and architecture between the cloud and the edge platforms.

10 TRADE-OFF BETWEEN FINE-TUNING ON THE EDGE AND IN THE CLOUD

Models can be fine-tuned on the cloud, but sending updated weights back from the cloud to the edge device has a higher power consumption. A solution for this is to fine-tune algorithms on the edge, thus potentially decreasing power consumption. The final question in this research concerns the optimization of the performance of a deployed edge device with the help of fine-tuning.

What is the trade-off between fine-tuning on the edge and in the cloud?

To investigate the difference in classification performance between fine-tuning on the edge and the cloud, the models are fine-tuned on images on the edge device. The classification heads of the edge compiled models are removed, after which they are retrained by the Last Layer Randomized weights Training (LLRT) method, as described in Section 9.1.2, for fine-tuning on the edge TPU.

The power consumption of the edge device is measured while fine-tuning the models. This can be used to compare the power consumption of fine-tuning the model on the edge with the power consumption of receiving updated weights when a model is fine-tuned on the cloud. These values are retrieved from the theoretical values as found in Section 7.2. This comparison leads to a trade-off between power consumption and classification accuracy after fine-tuning.

10.1 METHODOLOGY

The different steps needed for fine-tuning in the cloud and on the edge are illustrated in Figure 23.

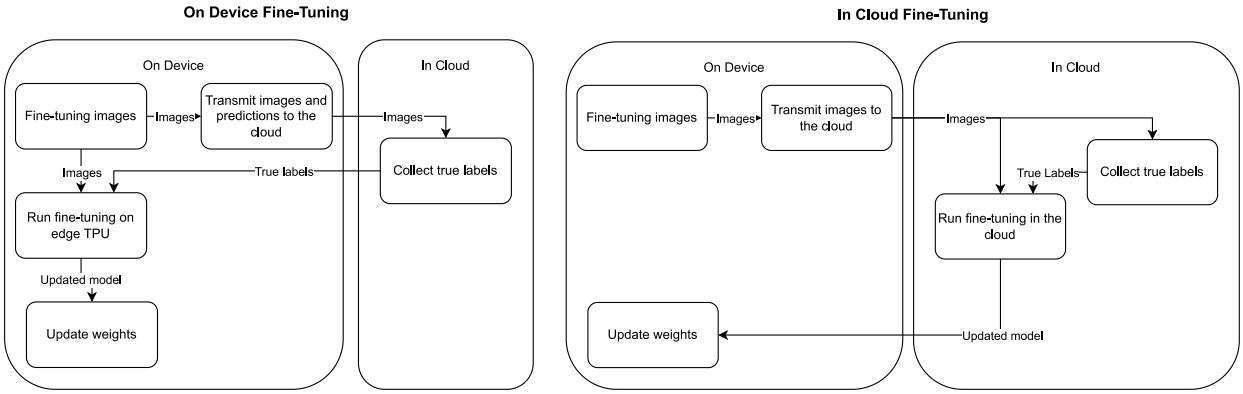


Figure 23: The different steps between fine-tuning on-device and fine-tuning in the cloud. The steps that happen on the edge device are the power-consuming parts which need investigation.

In Figure 23, the difference in power consumption between cloud and edge fine-tuning, is the running of fine-tuning on the edge, or the receiving of updated weights from the cloud. For the All Layers Small Learning Rate (ALSLR) [179] cloud fine-tuning method, the weights of the entire model need to be sent back, as opposed to the Last Layer Randomized weights Training (LLRT) [177] and Last Layer Small Learning Rate (LLSLR) [179] methods, where only the weights of the last layer need to be sent. Of these fine-tuning methods, which are described in Section 9.1.2, only LLRT is available for the edge.

10.1.1 *Model Fine-tuning*

The method for fine-tuning and evaluation, as shown in Figure 21, is the same for the edge. For fine-tuning models on the edge, the Google Coral development board offers two methods, weight imprinting [180] and backpropagation [181]. The backpropagation method was chosen as the weight imprinting method requires models that are pre-made for the weight imprinting method. The backpropagation method removes the last layer, where the model returns an image embedding of a vector of length 1280.

This vector can then be used as an input to a softmax layer, which is fine-tuned and runs on the CPU instead of the TPU, increasing classification time marginally. The fact that the last softmax layer is executed on the CPU instead of the TPU is a drawback of the backpropagation method [181].

10.1.2 *Model Evaluation*

The classification performance was measured the same as the previous experiment, as shown in Figure 21. Additionally, the power consumption of the edge device and time taken per fine-tuning method were measured, to evaluate the different ways of fine-tuning the deployed model. These measurements are done in the same way as described in Section 8.1.2.

Next to that, theoretical values for the differences in transmission were calculated. This entails the receiving of the updated model and weights for on-cloud fine-tuning and the power consumption values for receiving the true labels for every image on the edge fine-tuning. As the sending of images is needed both for edge and cloud, they will not be taken into account in the results of Section 10.2.2, but they will be taken into account in the results in Section 10.2.4.

10.1.3 *Data Size and Compression*

In the previous experiments, it was found that power consumption was mostly influenced by transmission time and thus the size of the data. To give some insights into viable options for reducing the size of the images which need to be sent, a pragmatic approach is taken to show the difference in the quality of the image and the image size. For this, three scenarios of data size and compression were taken into account. This was because of the ease of implementation of the techniques presented in this research. These compression techniques are:

- Image Compression Techniques: These are techniques to compress the images which are sent for feedback and fine-tuning.
- Image Embeddings: This would mean that only image embeddings are sent, which can later be reconstructed into an image. Image embeddings can also be used in a teacher-student configuration, where a teacher model is deployed in the cloud.
- Compression through the deployed ML model: The Machine Learning model, which is deployed on the edge device, already contains a way of data compression. Every layer compresses an input image further, thus these activation layers can be used to send compressed images to the human expert.

To see if the classes can be identified when the images are compressed, the compressed image can be compared to the original image. The compression ratio in Equation 10.1 is used to describe how much an image is compressed.

$$\text{Compression Ratio} = \frac{\text{Compressed Image Size}}{\text{Original Image Size}} \tag{10.1}$$

10.1.4 *Battery Charge over Time*

To illustrate the typical power draw over time of a deployed edge device, a model for battery charge and power usage can be used. This model uses the remote evaluation and fine-tuning methods added and incorporates the values of Chapter 7. This model estimates how long a deployed edge device can operate, given the inputs:

- Battery capacity: The full capacity of the battery used in the edge system.
- Inferences per day: The number of images that are inferred per day. In the case of the cloud, how many images are sent. In the case of the edge, how many images are inferred on the edge.

- Occurrence of remote evaluation and fine-tuning: How many times per a certain time period the system is remotely evaluated and fine-tuned.
- Fine-tuning method: The fine-tuning method that is used (LLRT, LLSLR, ALSLR), whether on the edge or on the cloud.
- Incoming charge: An average incoming charge from an external source.
- Image size: The size of the images which are sent to the cloud.
- Nominal and idle power usage: The power which is consumed during operation and during down-time, as well as the uptime.

10.1.5 *Experiment*

The fine-tuned models on the edge were prepared to be fine-tuned by the backpropagation method available to the Google Coral TPU development board. The last layer was removed from the model and then converted to a TF lite model and compiled as an edge TPU model. This model was then used together with the API given by Google Coral to retrieve the image embeddings from these models. These image embeddings were then used with the API [181], to train a new softmax layer on the TPU, which was executed on the CPU to then infer images. The evaluation sets were then used to finally retrieve the classification performance of the fine-tuned model. During the retrieval of image embeddings and the fine-tuning process, power measurements and time measurements were taken.

10.2 RESULTS

The focus of this experiment was to track the differences in classification performance and the differences in power consumption when deploying cloud fine-tuned models and edge fine-tuned models on the edge. These two evaluation metrics are discussed separately. Furthermore, a pragmatic approach has been taken to show the impact of data compression to investigate its viability. Finally, an example is given for a trade-off between the lifetime of an edge device, and its classification performance.

10.2.1 *Classification Performance*

The classification performance difference of the edge fine-tuning method compared to the cloud fine-tuning methods is shown in Figure 24. In this graph, only EfficientNetV2B0 is shown because this model performed the best, like in the previous experiment as shown in Section 9.2. The results for the other models are located in Appendix D and shown in Table 25 and Figure 53.

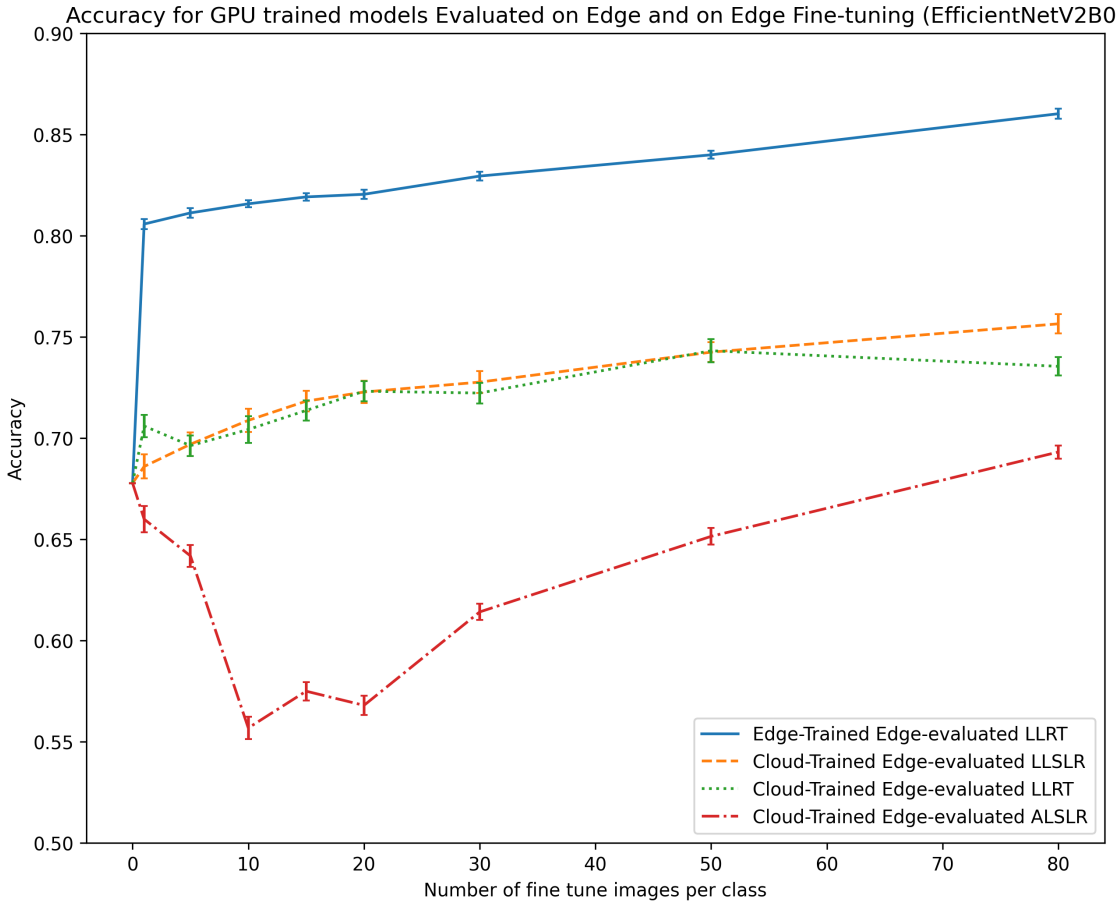


Figure 24: The results for EfficientNetV2B0 fine-tuned models that are trained on the cloud and tested on edge and fine-tuned on the edge. The accuracy is given for the varying amounts of fine-tune training images. The accuracy of 0 images per class is given by the non-fine-tuned models as given in experiment 3 in Section 8.2.

The results in Figure 24 and 53, show that the edge fine-tuned models perform significantly better than the cloud-trained models. There is an increase of 10% in accuracy over the best-performing edge-deployed cloud fine-tuned model.

The classification performance increases by 5% when increasing the number of images per class used for fine-tuning from 1 to 80. However, as shown in Figure 22, for one image per class, there is an increase of almost 15%, which was also seen in the previous experiment.

10.2.2 *Power Consumption for Fine-Tuning*

Figure 25 shows the time needed for fine-tuning the different models on the Google Coral development board is shown. The larger models take longer to train, mainly because retrieving the image embedding takes longer.

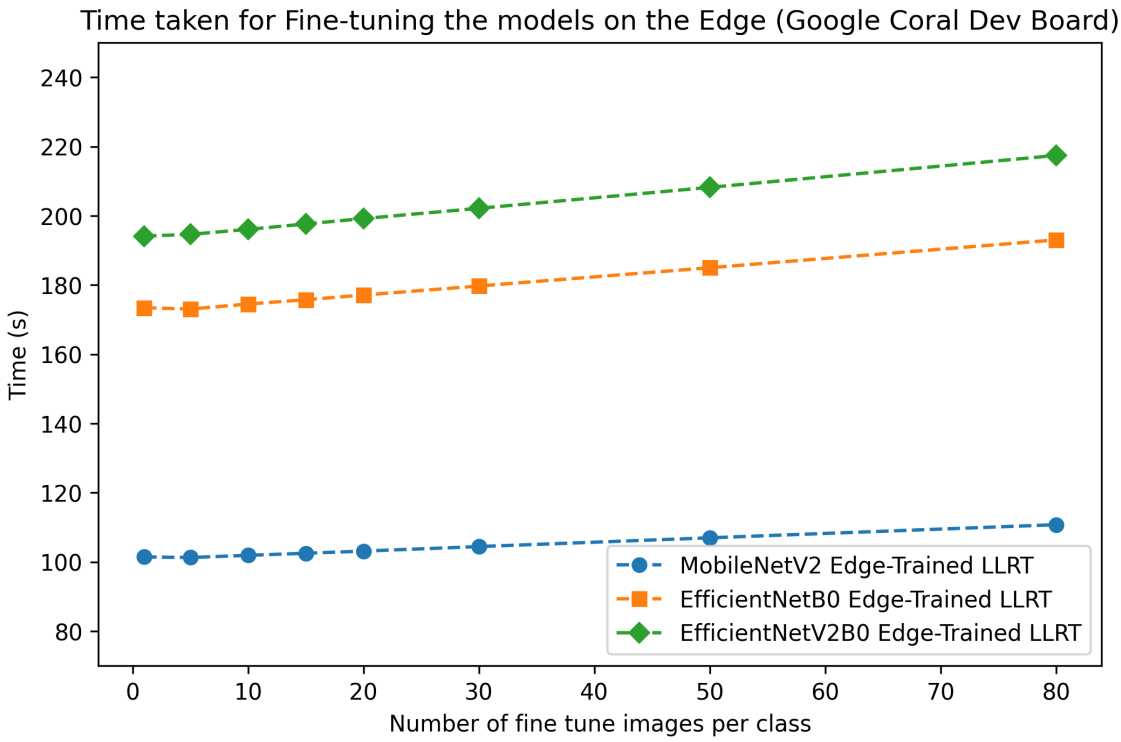


Figure 25: Time taken for fine-tuning per edge model in seconds.

Figure 26 shows the power consumption for fine-tuning each model on the edge device is shown. Table 18 shows the total power needed for a model to be fine-tuned for different fine-tuning methods and locations. The complete power consumption figures are shown in Appendix D.2, Figures 54, 55 and Tables 26, 27 and 28.

Figure 27 shows the power that is consumed when fine-tuning the different models. Out of the four models, three are trained on the edge and one on the cloud. All models are evaluated on the most efficient transmission protocol. The power consumption of the last cloud-based fine-tuning method is shown in Appendix D.2, in Figure 55. This was not shown here, due to the large difference in power consumption, resulting in an unclear figure.

Table 18: The power consumption of fine-tuning different models on the edge device, as well as the power needed to receive the updated weights from the cloud. These values are also shown in Figure 55.

Fine-Tuning Method	Power (J/model)	Extra Power (J/image)
Edge LLRT MobileNetV2	346.68	0.797
Edge LLRT EfficientNetB0	618.83	0.954
Edge LLRT EfficientNetV2B0	716.26	1.486
Cloud LLRT/LLSLR NB-IoT	728.64	0
Cloud ALSLR NB-IoT	16934.94	0

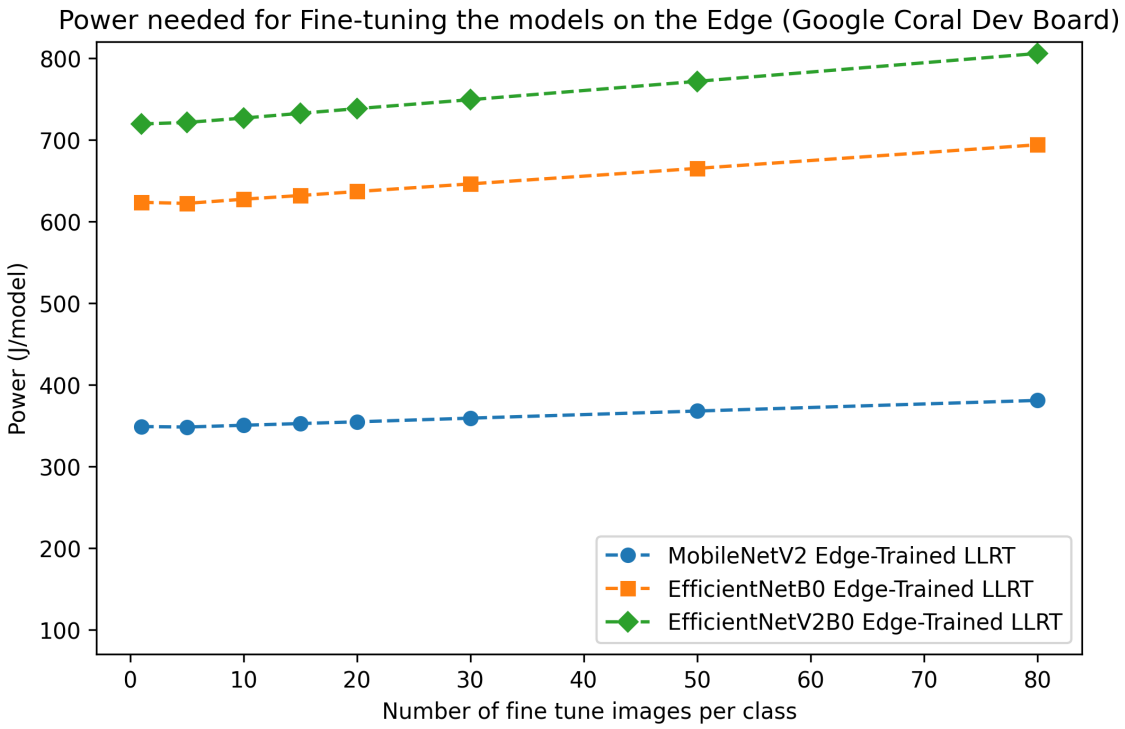


Figure 26: The power consumption of fine-tuning the edge models in Joule per model.

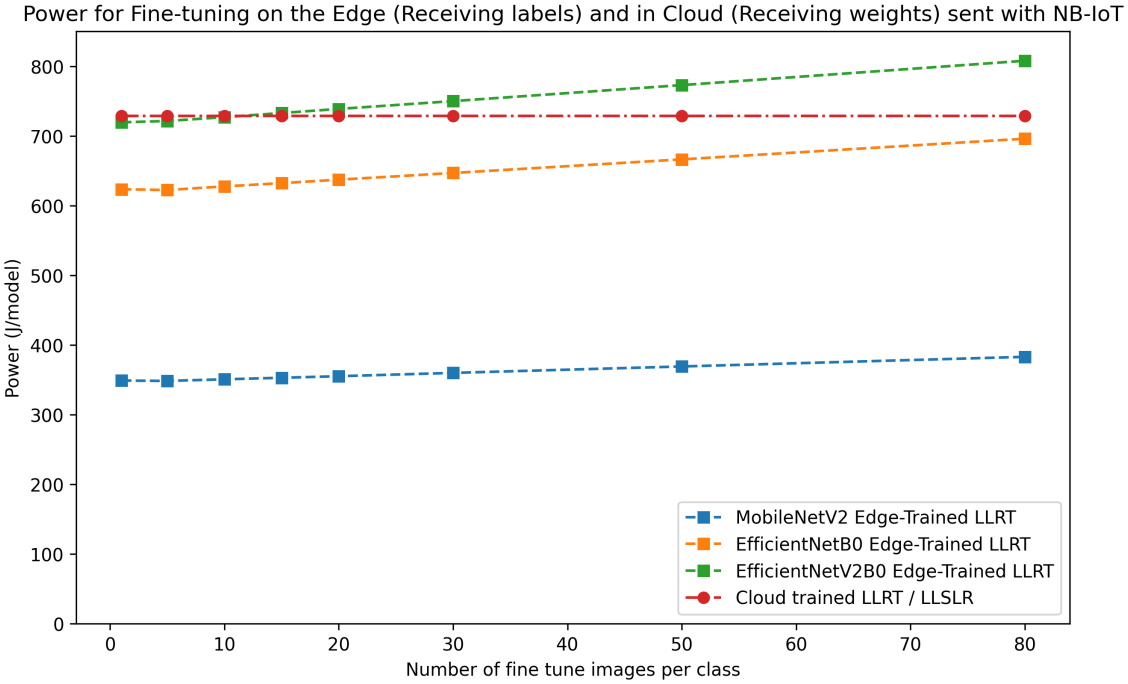


Figure 27: The power consumption of fine-tuning the edge models in Joule per model. The most power-efficient cloud model and transmission protocol are shown (LLRT and LLSLR, no ALSLR).

When comparing edge fine-tuning methods to cloud fine-tuning methods, it is seen that the power per model in this research is much higher for cloud-based training when compared to edge-based training, as seen in Table 18.

The only exception is when comparing the least power-intensive way of cloud fine-tuning and the most power-intensive edge-based fine-tuning for the largest model, EfficientNetV2B0, which is shown

in Figure 27. When up to 8 images per class were used for fine-tuning the EfficientNetV2B0 model, the edge fine-tuning methods consume less power. After 8 images per class, it is more power-efficient to use cloud-based fine-tuning.

10.2.3 *Data Size and Compression*

The pragmatic approach to data size and compression of data is to look at examples from the dataset, and examine how these images would look when subjected to different compression techniques and if the object is still discernible to human experts. In the dataset in this thesis, most of the different classes and objects are very distinguishable. However, when data compression is attempted for fine-grained datasets, the results can be different than those shown here.

For this example, the same images were used, as shown in Figure 12. Three good examples are the teddy, the remote and the phone, as one is very clear, and the other two can be hard to distinguish. The original images and compression are Figures 28 to 38.

Although the quality of the images is notably worse, the teddy can still be recognized in all of its images. For the remote and the phone, this is more difficult. This is mostly due to their unrecognisable colour, and their similarity to each other, as can be seen when comparing Figure 31 to Figure 38.

Figure 28: Original Teddy image scaled to 224 x 224 pixels (36.0KB).

Figure 29: Teddy image with 15 of 224 Principal Components (5.35KB).

Figure 30: Teddy image with 6% JPG compression (2.51KB).

Figure 31: Original Remote Image scaled to 224 x 224 pixels (32.0KB).

Figure 32: Remote image with 15 of 224 Principal Components (2.86KB).

Figure 33: Remote image with 6% JPG compression (1.78KB).

Figure 34: Remote image with 50 of 224 Principal Components (9.52KB).

Figure 35: Remote image with 22% JPG compression (4.47KB).

Figure 36: Original Phone Image scaled to 224 x 224 pixels (54.3KB).

Figure 37: Phone image with 50 of 224 Principal Components (28.0KB).

Figure 38: Phone image with 22% JPG compression (4.41KB).

Figures 39 to 41 shows the different activation layers for the EfficientNetV2B0 model, showing that image compression can be done with the already deployed ML models. The images are recognizable in the first hidden layer, and progressively get more compressed and become more difficult to recognize. PCA and JPG show similar compression results, in that the teddy seems to be recognizable up until the fifth hidden layer, whereas the difference between the remote and the phone is harder to recognize in itself. The fifth layer exists of 56 by 56 INT8 pixels, which results in 3136 pixels, thus 3.14KB.

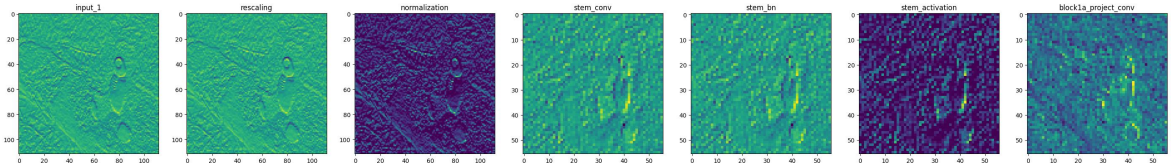


Figure 39: Activation layers from the neural network used (EfficientNetV2B0) for the teddy image.

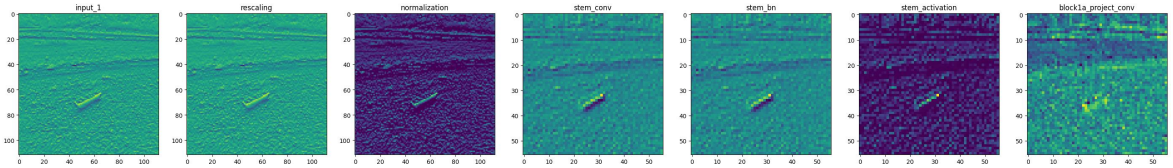


Figure 40: Activation layers from the neural network used (EfficientNetV2B0) for the remote image.

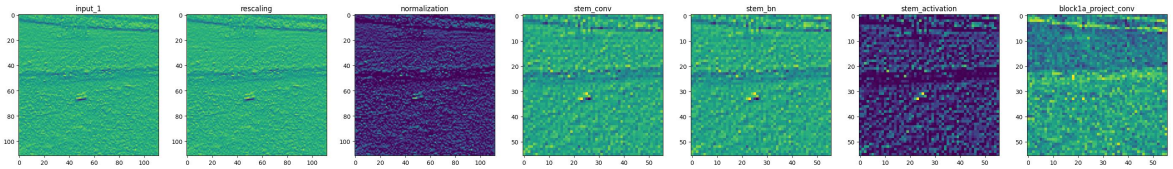


Figure 41: Activation layers from the neural network used (EfficientNetV2B0) for the phone image.

Table 19 shows the compression rate and the size of the compressed image, as compared to the original image of 36.0KB. The compression techniques can compress the original image to 2.51KB and are the best in compression to images that are still recognizable by human experts. If one uses the image embeddings for labelling, then the size of a 36.0KB image can be compressed further into 1.28KB.

Table 19: Results of compression in size and percentage. The percentages are calculated with an original image size of 36.0KB.

Compression technique	Compressed size (KB)	Compression (%)
Compression Techniques (JPG)	2.51	6.97
Image Embedding (Teacher-Student)	1.28	3.56
Activation Layers	3.14	8.72

10.2.4 *Battery Charge over Time*

Figure 42 visualizes the different ML models and different transmission protocols in a real-life application. This graph shows the potential battery charge curve of an edge device, which is evaluated and fine-tuned every month. The battery charge is chosen the same way as in experiment 2 in Section 7.2.2. The nominal and idle power usage and the uptime of the edge device are also chosen in the same way. In this example, 80 images per class are used for fine-tuning the models. In the graph, one can see the steps every 31 days,

as an evaluation set is sent, as well as the fact that the models are fine-tuned. 31 days were chosen, to be able to counteract the drift which occurs due to the changing of the seasons, while minimizing power consumed.

To give a comparison between different scenarios, the transmission protocol and fine-tuning method are varied. Figure 42 shows that ALSLR fine-tuning on the cloud leads to a significantly shorter battery life of the EI device when compared to LLRT and LLSLR on the cloud.

When comparing both edge fine-tuned options, the difference in transmission protocol leads to a reduced battery life. When comparing the cloud-based fine-tuning LLRT method to the edge-based fine-tuning LLRT method, the edge-based fine-tuning uses more power when fine-tuning for 80 images per class. Figure 42 shows the trade-off between power consumption and the different models, as fine-tuning the ML models on the edge leads to the edge device having 4 days less battery life. Fine-tuning on the edge also lead to a higher accuracy then fine-tuning on the cloud and deploying that model on the edge.

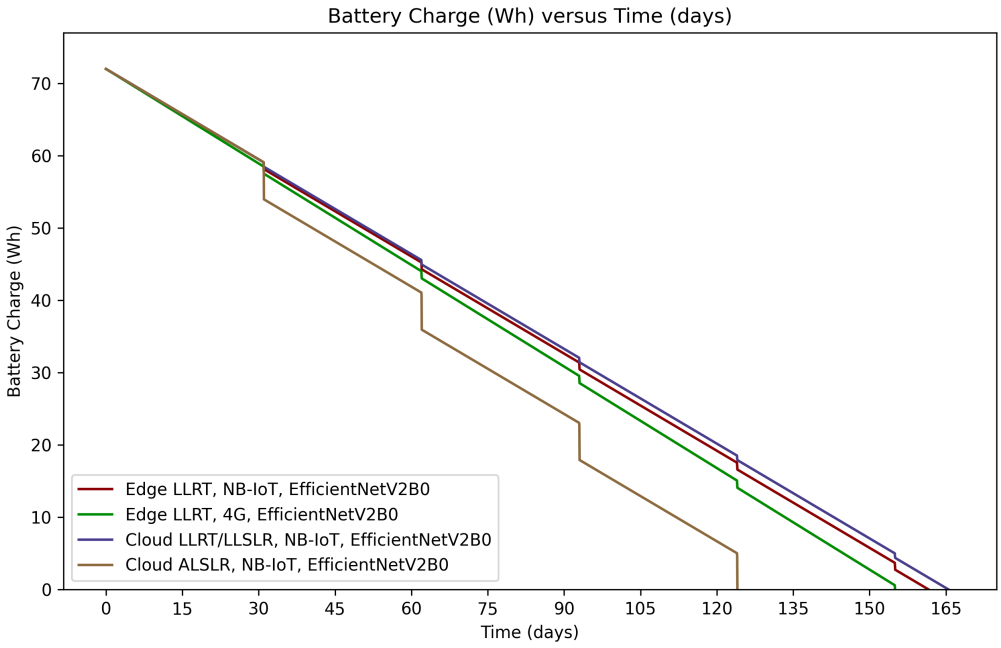


Figure 42: The battery charge of a 72Wh battery, when 100 36.0KB images are inferred every day, a 0.21% on time with 3.9W nominal power usage, and 0.001W idle power usage, 0W incoming power. Furthermore, 20 images per class are sent for evaluation, and 80 images per class for fine-tuning, for a total of 10 classes. The evaluation set, as well as fine-tuning, occurs every 31 days. In this graph, two different combinations of models and transmission protocols are shown.

10.3 DISCUSSION

There is an increase of 10% in classification performance of the edge fine-tuned models over the cloud fine-tuned models. This is due to the high increase in accuracy after only one image per class is used for fine-tuning on the edge. Besides that, the cloud fine-tuned models have a minimal increase in classification performance when deployed on the edge. If the classification performance drop when moving from the cloud to the edge is addressed, this could lead to the cloud and edge having similar performance or the cloud having higher classification performance.

The increase in accuracy for fine-tuning on 1 image per class was also seen in Section 9.2 for the LLRT cloud fine-tuned models. This suggests that the LLRT fine-tuning method works well for increasing the classification performance with the real-world dataset which is used.

A crossover point can be approximated where fine-tuning a model on the edge consumes more power than fine-tuning in the cloud. After this crossover point, it is more power-efficient to fine-tune the model in the cloud. This is due to the fact that the edge fine-tuned models are dependent on the number

of images per class, and the cloud is not. This crossover point would be at approximately 480 images per class for MobileNetV2, for EfficientNetB0 at approximately 115 images per class and for EfficientNetV2B0 at 8 images per class.

The results of the power consumption, as shown in Section 10.2.2, are not impacted by the compression of images. Data compression has shown promising results, as images could be compressed to 6.97% of the original size, while still being recognizable by a human. This can be reduced further to 3.56% to only an image embedding. Such image embeddings can be used by a teacher-student ML model, but an image embedding will not be recognizable by a human. It should be taken into account that some classes were similar to each other, as illustrated in Figure 31 until Figure 38. This means that for some classes, the distinction between these classes will be even more difficult when missing high-level details. It must be noted that even uncompressed images could be difficult to correctly label and identify for a human expert when dealing with fine-grained classes or classes being similar to each other.

As Figure 42 shows, there are a lot of different parameters that influence the lifespan of an edge device. These different parameters can all be adjusted, and the tool made in this experiment can be used to visualize how large the impact of a change in these input parameters can be, as well as see how some input parameters influence each other. The changing of these parameters could also influence the final choice between fine-tuning on the edge and in the cloud for certain edge devices.

The difference of 4 days lifespan between edge and cloud fine-tuning, poses a trade-off with the 10% accuracy increase that edge fine-tuned models bring with them. As 4 days compared to a lifespan of 165 images is negligible, it shows that edge-based fine-tuning is the preferred method of fine-tuning.

A limitation of this experiment is the scalability of the solution that is posed. When this system is deployed, a human expert would still need to label every image that is sent from the EI edge devices to the cloud. Even when the number of images that need to be checked for every edge device is small, a lot of deployed devices still lead to a lot of images that need to be checked by a human expert. When a student-teacher model would be employed, the human effort needed would decrease drastically, but this could influence accuracy, as a teacher model could misclassify the evaluation image, leading to incorrect fine-tuning.

10.4 CONCLUSION

The classification performance for models that are fine-tuned on the edge is at least 10% higher than for cloud fine-tuned models. For MobileNetV2, this ranges from 6% for a small number of fine-tuning images to 10% for around 80 fine-tuning images per class. For EfficientNetV2B0, this increase is about 10%, regardless of the number of images used for fine-tuning. For EfficientNetB0, this is even more as it is around 17%, regardless of the number of fine-tuning images.

Fine-tuning on the edge leads to less power consumed and higher classification performance on the edge for MobileNetV2 and EfficientNetB0. This holds for MobileNetV2 when using less than 480 images per class, and EfficientNetB0 for less than 115 images per class. For MobileNetV2, this is approximately 350 Joule per fine-tuning of the model, and for EfficientNetB0, this is approximately 100 Joule per fine-tuning. For both, this changes with how many images per class are used for fine-tuning the models.

For the largest model, EfficientNetV2B0, the power consumption of fine-tuning on a few images per class, is comparable to fine-tuning in the cloud. Comparing the classification performance then decides the trade-off, as the edge fine-tuned model performs a lot better than the cloud fine-tuned model.

Combining the fact that the edge-based methods for fine-tuning consume less power for fine-tuning while having a higher classification performance, the choice for edge-based fine-tuning is clear.

Furthermore, data compression could be a powerful method, to compress the images that are sent back, and possibly to send the labels or the updated ML model. For the images used in this research, it shows that some classes can be easily compressed without missing information, but for other classes, this is not the case.

A tool was made that shows the impact of different parameters on each other and on the lifetime of the deployed edge device. The presence of remote evaluation, as well as fine-tuning, can be seen in the sudden decrease in battery capacity, as well as the fact that some fine-tuning options lead to more power being consumed.

11

CONCLUSION AND FUTURE WORK

In this research, we investigated the reliable evaluation of deployed edge devices, the power consumption of the different components of a deployed edge device and the improvement in classification performance of a deployed edge device. The focus of this research was the impact of Out-of-Distribution (OOD) data and how real-life data influences the evaluation and improvement of classification performance. In this chapter, the conclusions that are drawn in this research are presented, as well as future work and limitations to be adressed.

11.1 CONCLUSIONS

Edge Intelligence (EI) devices have gained large interest in research in previous years, due to their ability to reduce the large amounts of data which are generated by edge devices. A lot of research in deploying these devices and their Machine Learning (ML) models has already been done. Despite this interest, research has been limited with regards to post-deployment monitoring of these devices, their ability to handle Out-of-Distribution (OOD) data, and to improve upon their classification performance with this knowledge.

The main focus of this research was to solve these open issues, which discuss post-deployment monitoring of EI devices and the OOD nature of real-world data while investigating the possibility of increasing classification performance while minimizing power consumption. This research could improve the scalability of edge devices, as well as give insight into where fine-tuning may occur to improve classification performance while minimizing power consumption. To this end, the main question to be answered by this research was formulated as follows:

> ***How to achieve the best classification performance and reliable evaluation for a deployed edge intelligent device while using the least amount of power?***

To answer this question and conclude this research, this main research question is addressed by answering the five sub-questions.

> *How to make a standardised edge AI distribution shift dataset that allows supervision over different distributions?*

A real-world dataset is presented, comprising different domains to study the OOD nature of real-life data. This dataset consists of six domains, in which ten different classes were placed for a total of 100 images per class per domain. This resulted in a dataset of 6000 images. This dataset is easily extendable by ensuring that the domains in which the objects are placed, and the objects themselves are accessible.

> *How many images are needed, for a statistically reliable evaluation of a multi-class classification?*

This research has shown with 95% confidence that the maximum Confidence Interval would be 2% for 15 to 25 images per class. This indicates that around 15 to 25 images per class can be used for a reliable evaluation of EI devices. Furthermore, the EfficientNet models outperformed IncepctionResNetV2 with a 10% difference in accuracy across all distributions and MobileNetV2 with a 10 to 20% difference in accuracy across all distributions. The domains most interesting for classification performance are the uniform distribution, which generally performed the best, and the park distribution, which generally performed the worst.

> *What is the trade-off between evaluation reliability, power usage, and human effort?*

For the evaluation of EI edge devices, images and predictions need to be sent back to evaluate the accuracy of the edge device by a human expert. This research identified the transmission of images to be the most power-consuming component of the power-consuming components of an edge device.

NB-IoT is the most power-efficient with 198 J/image, with 4G being the second most power-efficient with 217.0 J/image. It was found that with the NB-IoT protocol, about 36400 36.0 KB images

can be sent in about 65900 seconds before a 72Wh battery is drained with no incoming power. As around 150 to 250 images are needed for a reliable evaluation, a good evaluation can be attained with these settings. All protocols can send enough images for a reliable evaluation set, but some could send a lot more evaluation sets than others.

Reducing human effort was done by reducing the number of images needed for a reliable evaluation. To even further reduce human effort, a student-teacher approach can be taken by employing a larger, more accurate cloud model to label the evaluation images and check the predictions of the edge model.

What is the trade-off between inference on edge CPU or edge TPU, and how does quantisation influence this?

When comparing classification performance, EfficientNetV2S was the best in classifying the classes of the OOD distribution with 10 to 25% better accuracy than the other models. However, it could not be compiled for the edge TPU. Therefore, in this research, EfficientNetV2B0 is the best model which could be run on edge CPU and edge TPU, using quantisation to fit the model. EfficientNetV2B0 was 8 to 15% better in classifying OOD distribution classes as compared to EfficientNetB0, InceptionResNetV2 and MobileNetV2. Furthermore, a significant drop in classification performance was observed when running the quantised TF lite model in the cloud compared to on edge. This drop varies from about 7 to 16% between cloud and edge models, with the cloud models performing better. This drop was smaller for EfficientNetV2S and EfficientNetV2B0, and bigger for MobileNetV2 and EfficientNetB0. This difference in performance could be due to varying factors. For one, different hardware is used, which means that different hardware mapping was used, thus influencing the performance. Due to the different hardware, also different libraries needed to be used for this hardware. Although these libraries are officially supported libraries, they may still interpret code differently. Lastly, there could be optimisations in check for the different models when run on cloud hardware, which was not present for the edge, thus also influencing the final performance.

Finally, the experiments have shown that the smaller models consume less power. For example, MobileNetV2 consumes the least power while inferring captured images, with EfficientNetB0 and EfficientNetV2B0 following. On the other hand, EfficientNetV2S and InceptionResNetV2 both significantly increase power consumption, due to an increase in inference time.

What is the trade-off between inference of cloud fine-tuned models on cloud and edge, and how does quantisation influence this?

Similar to inferring the non-fine-tuned models, when inferring the fine-tuned models on the edge, the classification performance is significantly worse, with a difference of about 25%. This difference can be attributed to the same reasons as the previous experiment, as the same difference in hardware and software is present in this experiment. From this similarity in performance drop, it can be concluded that the performance drop is not caused by the way how the model is trained but may be caysed by how the model is quantised and compiled for the edge device. Moreover, the drop could also be caused by the difference between libraries employed on the edge as compared to in the cloud.

What is the trade-off between fine-tuning on the edge and in the cloud?

Finally, the classification performance difference when fine-tuning on the edge as compared to in the cloud is examined. Overall, all models performed better when inferring on the edge when fine-tuned on the edge than when fine-tuned on the cloud. The difference in classification accuracy was about 10% for EfficentNetV2B0 when compared to the best-performing cloud fine-tuned model (Last Layer Small Learning Rate (LLSLR)). EfficientNetB0 achieved similar performance, increasing performance over the best-performing cloud fine-tuned model (All Layers Small Learning Rate (ALSLR)) with over 17%.

Worth noting is that even with 1 image per class, classification performance increased with the same percentages as mentioned above. An example of this is the performance of EfficientNetV2B0, rising from 67.8% accuracy with no fine-tuning to 80.6% with just 1 image per class, and reaching a final accuracy of 86.0% with 80 images per class. From these results, it can be seen that with 1 image per class of fine-tuning on an edge device, performance can increase massively without consuming a lot of power

Fine-tuning on the edge is, in most cases, worth the trade-off of performance and power consumption. It is significantly more power-efficient to fine-tune and receive labels for the images used in fine-tuning than to receive entire models or just the last layer in weights. This is due to the fact that label files are smaller than weights and biases files, thus decreasing the amount of energy needed for receiving these files. From the results that are presented, a trend line can be calculated. When fine-tuning on the edge, MobileNetV2 is faster to fine-tune, thus reducing power consumption, whereas EfficientNetV2B0 is the slowest model to fine-tune, thus increasing power consumption. When continuing the trend line for power consumption for the MobileNetV2 model, it can be seen that one would need to fine-tune about 480 images per class for Last Layer Randomized weights Training (LLRT) or LLSLR cloud fine-tuning to be viable. For the EfficientNetV2B0 model, one would need to fine-tune on 8 images per class before using LLRT or LLSLR cloud fine-tuning would be a viable alternative. As already 1 image per class makes a large difference in classification performance, fine-tuning on the edge would be suitable for all models presented here. When using more images per class, the difference in lifespan is approximately 4 days for a total of 165 days, but the trade-off of this is an increase of 10% in accuracy.

Overall, the performance of the edge fine-tuned models is better than that of the cloud fine-tuned models. However, if the drop-off in cloud fine-tuned models deployed on the edge is mitigated, an actual trade-off between accuracy and power can be made. When power consumption is less of a problem, and one optimizes for performance, EfficientNetV2B0 is a viable option, as it classifies the best, but also requires more power consumption. Lastly, when fine-tuning for more than 8 images per class, the LLRT or LLSLR cloud fine-tuning techniques are more power efficient.

How to achieve the best classification performance and reliable evaluation for a deployed edge intelligent device while using the least amount of power?

This research has shown the answer to this main research question in smaller steps. To be able to achieve the best classification performance, first, the deployed device needs to be reliably evaluated. This can be done with around 15 to 25 images per class. This reduction in images leads to reduced power consumed for the transmission of these images to the cloud.

For the transmission of these images, it is best to use either NB-IoT or 4G as a radio to send these images. These radio protocols have the smallest power consumption per data size, leading to a reduction in power consumption over other protocols like LTE-M or Wi-Fi.

The trade-off for achieving the best classification performance for a ML model run on an edge device was divided over multiple experiments. These experiments have shown that when deploying a cloud-trained model, fine-tuned or not, the classification performance is significantly impacted. Fine-tuning can be used to increase the classification performance of ML models deployed on the edge. For the best classification performance for the investigated ML models, edge-based fine-tuning should be used. For fine-tuning the model, a small number of images per class is needed for a relatively high increase in classification performance, allowing for low power consumption for a large performance increase.

11.2 FUTURE WORK

The first experiment showed models that were pre-trained on ImageNet. The ImageNet dataset is extensive but is not focussed on the handling of OOD data. The COCO dataset would be more suitable for pre-training the images on different classes and environments, due to the inherent OOD nature of the images, next to the overlap in classes. The images of the COCO dataset often have multiple objects in the image, so they would need to be chosen such, that there would be no other objects in the image, as this could lead to wrong classification.

The second experiment mainly used theoretical values provided by manufacturers or widely used networks. The power consumption model can be improved by performing real-world measurements. To do these measurements, files with different data sizes can be sent with the transmission protocols of interest. Besides varying file sizes, different modules for every transmission protocol can be investigated, as different modules may have varying amounts of power consumption. From these measurements, a graph can be made, to check if the power consumption of the different protocols is linear to the data size. Furthermore, such a graph would show the ideal transmission protocol and accompanying module for a

certain file size. Likewise, it is interesting to further investigate compression techniques to reduce image size and, thus, the power needed for sending and receiving data.

The results from the third experiment show a drop in performance when moving the TF lite model from the cloud to the edge. In the discussion of the experiment, possible reasons were already discussed, these being the different used libraries, the other hardware architecture, or optimisation for the cloud instead of the edge. These various reasons would need to be investigated to see which adjustments would improve the accuracy of the deployed and fine-tuned models running on the edge.

Eventually, it would be interesting to fully automate the learning of the ML models by using a big cloud-based model to give the true label to the images used for fine-tuning, which would not need to be done by a human expert. A bigger model would then lead to a teacher-student model structure, as attempted on a smaller scale in the work of de Prado [13]. This would also lead to less human effort needed to operate an extensive system of deployed edge devices, leading to better scalability.

As Figure 42 shows, there are a lot of different parameters that influence the uptime of an edge device. These different parameters can all be adjusted, and the tool which has been made in experiment 2 and adjusted for experiment 5 can be used to visualize how large the impact of a change in these input parameters can be. This model could be made into an interactive tool to create insights for other people who have a system in how remote evaluation and fine-tuning could influence the lifetime of a deployed edge device.

Lastly, a limitation of this research is that the results shown are limited to the given dataset and problem statement. When another dataset is used, the specific results concerning the classification performance of the models may differ from what is presented in this research, which could alter the trade-off made in this thesis. The methods which are presented in this thesis may be followed to gather the classification performance, fine-tuning performance and power consumption of running ML models on edge devices for a different dataset than presented in Chapter 5.

BIBLIOGRAPHY

[1] SeedScientific. 'How Much Data Is Created Every Day? [27 Powerful Stats]'. URL: https://seedscientific.com/how-much-data-is-created-every-day/. *SeedScientific*.

[2] Daniel Price. 'Infographic: How much data is produced every day?' URL: https://cloudtweaks.com/2015/03/how-much-data-is-produced-every-day/. *CloudTweaks.com*.

[3] Subhas Chandra Mukhopadhyay et al. 'Artificial Intelligence-Based Sensors for Next Generation IoT Applications: A Review'. In: *IEEE Sensors Journal* 21.22 (Nov. 2021), pp. 24920–24932. ISSN: 15581748. DOI: 10.1109/JSEN.2021.3055618.

[4] Xiaofei Wang et al. 'Convergence of Edge Computing and Deep Learning: A Comprehensive Survey'. In: *IEEE Communications Surveys and Tutorials* 22.2 (Apr. 2020), pp. 869–904. ISSN: 1553877X. DOI: 10.1109/COMST.2020.2970550.

[5] Carlos Poncinelli Filho et al. 'A Systematic Literature Review on Distributed Machine Learning in Edge Computing'. In: *Sensors 2022, Vol. 22, Page 2665* 22.7 (Mar. 2022), p. 2665. ISSN: 1424-8220. DOI: 10.3390/S22072665. URL: https://www.mdpi.com/1424-8220/22/7/2665/htm%20https://www.mdpi.com/1424-8220/22/7/2665.

[6] Massimo Merenda, Carlo Porcaro and Demetrio Iero. 'Edge Machine Learning for AI-Enabled IoT Devices: A Review'. In: *Sensors 2020, Vol. 20, Page 2533* 20.9 (Apr. 2020), p. 2533. ISSN: 1424-8220. DOI: 10.3390/S20092533. URL: https://www.mdpi.com/1424-8220/20/9/2533/htm%20https://www.mdpi.com/1424-8220/20/9/2533.

[7] Unknown. '1. Embedded AI: bringing AI in the wild - Nederlandse AI Coalitie'. URL: https://nlaic.com/event-programma/1-embedded-ai-bringing-ai-in-the-wild/.

[8] Edge Impulse. 'What is embedded ML, anyway?' URL: https://docs.edgeimpulse.com/docs/what-is-embedded-machine-learning-anyway.

[9] Tiffany Yeung. 'What Is Edge AI and How Does It Work? | NVIDIA Blog'. URL: https://blogs.nvidia.com/blog/2022/02/17/what-is-edge-ai/.

[10] Zhi Zhou et al. 'Edge Intelligence: Paving the Last Mile of Artificial Intelligence With Edge Computing'. In: *Proceedings of the IEEE* (2019). ISSN: 00189219. DOI: 10.1109/JPROC.2019.2918951.

[11] Sam Leroux et al. 'TinyMLOps: Operational Challenges for Widespread Edge AI Adoption'. In: (Mar. 2022). DOI: 10.48550/arxiv.2203.10923. URL: http://arxiv.org/abs/2203.10923.

[12] Du Phan. 'A Primer on Data Drift.' URL: https://medium.com/data-from-the-trenches/a-primer-on-data-drift-18789ef252a6.

[13] Miguel de Prado et al. 'Robustifying the Deployment of tinyML Models for Autonomous Mini-Vehicles'. In: *Sensors (Basel, Switzerland)* 21.4 (Feb. 2021), pp. 1–16. ISSN: 14248220. DOI: 10.3390/S21041339. URL: /pmc/articles/PMC7918899/%20/pmc/articles/PMC7918899/?report=abstract%20https://www.ncbi.nlm.nih.gov/pmc/articles/PMC7918899/.

[14] Puranjay Mohan, Aditya Jyoti Paul and Abhay Chirania. 'A tiny cnn architecture for medical face mask detection for resource-constrained endpoints'. In: *Lecture Notes in Electrical Engineering* 756 LNEE (2021), pp. 657–670. ISSN: 18761119. DOI: 10.1007/978-981-16-0749-3{_}52/FIGURES/6. URL: https://link.springer.com/chapter/10.1007/978-981-16-0749-3_52.

[15] Dr Lachit Dutta and Swapna Bharali. 'TinyML Meets IoT: A Comprehensive Survey'. In: *Internet of Things* 16 (Dec. 2021), p. 100461. ISSN: 2542-6605. DOI: 10.1016/J.IOT.2021.100461.

[16] Partha Pratim Ray. 'A review on TinyML: State-of-the-art and prospects'. In: *Journal of King Saud University - Computer and Information Sciences* 34.4 (Apr. 2022), pp. 1595–1623. ISSN: 1319-1578. DOI: 10.1016/J.JKSUCI.2021.11.019.

[17] Karina Zadorozhny and Giovanni Cinà. 'Out-Of-Distribution Detection in Medical AI | by Pacmed | Geek Culture | Medium'. URL: https://medium.com/geekculture/out-of-distribution-detection-in-medical-ai-b638b385c2a3.

[18] Han Cai et al. 'Tiny Transfer Learning: Towards Memory-Efficient On-Device Learning'. In: (July 2020). DOI: 10.48550/arxiv.2007.11622. URL: https://arxiv.org/abs/2007.11622v2.

[19] Chip Huyen. 'Slides ML - cs329s_2022_10_slides_ml_failure_diagnosis'. URL: https://docs.google.com/presentation/d/1tuCIbk9Pye-RK1xqiiZXPzT8lIgDUL6CqBkFSYZXkbY/edit#slide=id.g112c1e99806_0_791.

[20] Neerav Karani. 'Tackling Distribution Shifts in Machine Learning-Based Medical Image Analysis'. In: (2022).

[21] Digiteum. 'Differences Between Cloud, Fog and Edge Computing | Digiteum'. URL: https://www.digiteum.com/cloud-fog-edge-computing-iot/.

[22] Arun. 'An Introduction to TinyML. Machine Learning Meets Embedded Systems | Towards Data Science'. URL: https://towardsdatascience.com/an-introduction-to-tinyml-4617f314aa79.

[23] Weixing Su et al. 'AI on the edge: a comprehensive review'. In: *Artificial Intelligence Review 2022* (Mar. 2022), pp. 1–59. ISSN: 1573-7462. DOI: 10.1007/S10462-022-10141-4. URL: https://link.springer.com/article/10.1007/s10462-022-10141-4.

[24] José Ángel et al. 'CEML: Mixing and moving complex event processing and machine learning to the edge of the network for IoT applications'. In: (2016). DOI: 10.1145/2991561.2991575. URL: http://dx.doi.org/10.1145/2991561.2991575.

[25] Qiang Yang et al. 'Federated machine learning: Concept and applications'. In: *ACM Transactions on Intelligent Systems and Technology* 10.2 (2019). ISSN: 21576912. DOI: 10.1145/3298981. URL: https://doi.org/10.1145/3298981.

[26] Yao Chung Chang and Ying Hsun Lai. 'Campus Edge Computing Network Based on IoT Street Lighting Nodes'. In: *IEEE Systems Journal* 14.1 (Mar. 2020), pp. 164–171. ISSN: 19379234. DOI: 10.1109/JSYST.2018.2873430.

[27] Jacob Wilhelm Kamminga. 'Hiding in the Deep: Online Animal Activity Recognition using Motion Sensors and Machine Learning'. In: (Sept. 2020). DOI: 10.3990/1.9789036550550. URL: https://research.utwente.nl/en/publications/hiding-in-the-deep-online-animal-activity-recognition-using-motio.

[28] Vidya Zope et al. 'TRAIL-TRACKER: ANTI-POACHING INTELLIGENCE USING AI AND IOT'. In: 8 (2020), p. 2063. URL: www.ijcrt.org.

[29] Thijs Suijten and Tim van Deursen. 'Protecting wildlife with machine learning'. URL: https://engineering.q42.nl/hack-the-poacher/.

[30] Jacob Kamminga et al. 'Poaching Detection Technologies—A Survey'. In: *Sensors 2018, Vol. 18, Page 1474* 18.5 (May 2018), p. 1474. ISSN: 1424-8220. DOI: 10.3390/S18051474. URL: https://www.mdpi.com/1424-8220/18/5/1474/htm%20https://www.mdpi.com/1424-8220/18/5/1474.

[31] 'What is a Machine Learning Pipeline? - Datatron'. URL: https://datatron.com/what-is-a-machine-learning-pipeline/.

[32] Pete Warden and Daniel Situnayake. *TinyML*. 2020, p. 504. ISBN: 9781492052036. URL: https://www.oreilly.com/library/view/tinyml/9781492052036/.

[33] Ken Hoffman. 'Machine Learning: How to Prevent Overfitting'. URL: https://medium.com/swlh/machine-learning-how-to-prevent-overfitting-fdf759cc00a9.

[34] Chip Huyen. 'Data Distribution Shifts and Monitoring'. URL: https://huyenchip.com/2022/02/07/data-distribution-shifts-and-monitoring.html#data-shifts%20https://huyenchip.com/2022/02/07/data-distribution-shifts-and-monitoring.html.

[35] Swapnil Sayan Saha, Sandeep Singh Sandha and Mani Srivastava. 'Machine Learning for Microcontroller-Class Hardware – A Review'. In: *eps IEEE SENSORS JOURNAL* XX (2022), p. 1. URL: http://arxiv.org/abs/2205.14550.

[36] 'University Library | Service Portal | University of Twente'. URL: https://www.utwente.nl/en/service-portal/university-library.

[37] 'Google Scholar'. URL: https://scholar.google.com/.

[38] Young Hyun Yoon et al. 'Intellino: Processor for Embedded Artificial Intelligence'. In: *Electronics 2020, Vol. 9, Page 1169* 9.7 (July 2020), p. 1169. ISSN: 2079-9292. DOI: 10.3390/ELECTRONICS9071169. URL: https://www.mdpi.com/2079-9292/9/7/1169/htm%20https://www.mdpi.com/2079-9292/9/7/1169.

[39] Errin O'Connor. 'Embedded Intelligence for (IoT) smart process and services - EPCGroup'. URL: https://www.epcgroup.net/embedded-intelligence/.

[40] Mahadev Satyanarayanan et al. 'The case for VM-based cloudlets in mobile computing'. In: *IEEE Pervasive Computing* 8.4 (Oct. 2009), pp. 14–23. ISSN: 15361268. DOI: 10.1109/MPRV.2009.82.

[41] Ion Stoica et al. 'A Berkeley View of Systems Challenges for AI'. In: (Dec. 2017). DOI: 10.48550/arxiv.1712.05855. URL: https://arxiv.org/abs/1712.05855v1.

[42] Shuiguang Deng et al. 'Edge Intelligence: The Confluence of Edge Computing and Artificial Intelligence'. In: *IEEE Internet of Things Journal* 7.8 (Aug. 2020), pp. 7457–7469. ISSN: 23274662. DOI: 10.1109/JIOT.2020.2984887.

[43] Fouad Sakr et al. 'Machine Learning on Mainstream Microcontrollers'. In: *Sensors 2020, Vol. 20, Page 2638* 20.9 (May 2020), p. 2638. ISSN: 1424-8220. DOI: 10.3390/S20092638. URL: https://www.mdpi.com/1424-8220/20/9/2638/htm%20https://www.mdpi.com/1424-8220/20/9/2638.

[44] Aleks Huč, Jakob Šalej and Mira Trebar. 'Analysis of Machine Learning Algorithms for Anomaly Detection on Edge Devices'. In: *Sensors 2021, Vol. 21, Page 4946* 21.14 (July 2021), p. 4946. ISSN: 1424-8220. DOI: 10.3390/S21144946. URL: https://www.mdpi.com/1424-8220/21/14/4946/htm%20https://www.mdpi.com/1424-8220/21/14/4946.

[45] H. Brendan McMahan et al. 'Communication-efficient learning of deep networks from decentralized data'. In: *Proceedings of the 20th International Conference on Artificial Intelligence and Statistics, AISTATS 2017*. PMLR, Feb. 2017. ISBN: 1602.05629v3. DOI: 10.48550/arxiv.1602.05629. URL: https://arxiv.org/abs/1602.05629v3.

[46] Wenyu Zhang et al. 'Client Selection for Federated Learning with Non-IID Data in Mobile Edge Computing'. In: *IEEE Access* 9 (2021), pp. 24462–24474. ISSN: 21693536. DOI: 10.1109/ACCESS.2021.3056919.

[47] Google. 'TensorFlow Lite | ML for Mobile and Edge Devices'. URL: https://www.tensorflow.org/lite.

[48] 'Convert TensorFlow models | TensorFlow Lite'. URL: https://www.tensorflow.org/lite/models/convert/convert_models.

[49] Mohammed Zubair M. Shamim. 'Hardware Deployable Edge-AI Solution for Pre-screening of Oral Tongue Lesions using TinyML on Embedded Devices'. In: *IEEE Embedded Systems Letters* (2022). ISSN: 19430671. DOI: 10.1109/LES.2022.3160281.

[50] Marco Giordano et al. 'Design and Performance Evaluation of an Ultralow-Power Smart IoT Device With Embedded TinyML for Asset Activity Monitoring'. In: *IEEE Transactions on Instrumentation and Measurement* 71 (2022), pp. 1–11. ISSN: 0018-9456. DOI: 10.1109/TIM.2022.3165816. URL: https://ieeexplore.ieee.org/document/9758676/.

[51] Pang Wei Koh et al. 'WILDS: A Benchmark of in-the-Wild Distribution Shifts'. In: (July 2020), pp. 5637–5664. ISSN: 2640-3498. URL: https://proceedings.mlr.press/v139/koh21a.html%20http://arxiv.org/abs/2012.07421.

[52] Baochen Sun and Kate Saenko. 'Deep CORAL: Correlation Alignment for Deep Domain Adaptation'. In: *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)* 9915 LNCS (July 2016), pp. 443–450. ISSN: 16113349. DOI: 10.1007/978-3-319-49409-8{_}35. URL: https://arxiv.org/abs/1607.01719v1.

[53] Martin Arjovsky et al. 'Invariant Risk Minimization'. In: (July 2019). URL: https://arxiv.org/abs/1907.02893v3.

[54] Shiori Sagawa et al. 'Distributionally Robust Neural Networks'. In: (Apr. 2020).

[55] Shiori Sagawa et al. 'Extending the WILDS Benchmark for Unsupervised Adaptation'. In: (Dec. 2021). URL: https://wilds.stanford.edu.%20http://arxiv.org/abs/2112.05090.

[56] Olivia Wiles et al. 'A fine-grained analysis of robustness to distribution shifts'. URL: https://arxiv.org/abs/2110.11328..

[57] Yuge Shi et al. 'How robust are pre-trained models to distribution shift?' In: (June 2022). DOI: 10.48550/arxiv.2206.08871. URL: https://arxiv.org/abs/2206.08871v1%20http://arxiv.org/abs/2206.08871.

[58] Sauptik Dhar et al. 'A Survey of On-Device Machine Learning: An Algorithms and Learning Theory Perspective'. In: (2021). DOI: 10.1145/3450494. URL: https://doi.org/10.1145/3450494.

[59] Bin Wang et al. 'A framework for self-supervised federated domain adaptation'. In: *Eurasip Journal on Wireless Communications and Networking* 2022.1 (Dec. 2022), pp. 1–17. ISSN: 16871499. DOI: 10.1186/s13638-022-02104-8. URL: https://link.springer.com/articles/10.1186/s13638-022-02104-8%20https://link.springer.com/article/10.1186/s13638-022-02104-8.

[60] QualComm. 'Snapdragon 8 Series Mobile Platforms'. URL: https://www.qualcomm.com/products/application/smartphones/snapdragon-8-series-mobile-platforms.

[61] HiSilicon. 'Kirin 990 Chipset | HiSilicon Official Site'. URL: https://www.hisilicon.com/en/products/Kirin/Kirin-flagship-chips/Kirin-990.

[62] HiSilicon. 'HUAWEI Ascend 910 Chipset'. URL: https://www.hisilicon.com/en/products/Ascend/Ascend-910.

[63] MediaTek. 'MediaTek Helio P60'. URL: https://www.mediatek.com/products/smartphones-2/mediatek-helio-p60.

[64] Google Cloud. 'Cloud Tensor Processing Units (TPUs)'. URL: https://cloud.google.com/tpu/docs/tpus.

[65] Intel. 'Intel® Xeon® D-2100 Processor Product Brief'. URL: https://www.intel.com/content/www/us/en/products/docs/processors/xeon/d-2100-brief.html.

[66] Samsung. 'Exynos 9820 | Mobile Processor'. URL: https://semiconductor.samsung.com/processor/mobile-processor/exynos-9-series-9820/.

[67] 'GrAI Matter Labs | Fastest Edge AI Processor'. URL: https://www.graimatterlabs.ai/.

[68] Baetyl. 'baetyl/baetyl: Extend cloud computing, data and service seamlessly to edge devices.' URL: https://github.com/baetyl/baetyl.

[69] Microsoft Azure. 'IoT Edge | Cloud Intelligence'. URL: https://azure.microsoft.com/nl-nl/services/iot-edge/.

[70] EdgeXFoundry. 'EdgeX'. URL: https://www.edgexfoundry.org/.

[71] NVIDIA. 'EGX Platform for Accelerated Computing'. URL: https://www.nvidia.com/en-us/data-center/products/egx/.

[72] Amazon Web Services. 'Intelligence at the IoT Edge — AWS IoT Greengrass'. URL: https://aws.amazon.com/greengrass/.

[73] Google Cloud. 'Google Cloud IoT - Fully Managed IoT Services'. URL: https://cloud.google.com/solutions/iot.

[74] 'OpenVINO™ Documentation — OpenVINO™ documentation — Version(latest)'. URL: https://docs.openvino.ai/latest/index.html.

[75] Sparkfun. 'SparkFun Edge Development Board - Apollo3 Blue - DEV-15170'. URL: https://www.sparkfun.com/products/15170.

[76] Jisu Kwon and Daejin Park. 'Hardware/Software Co-Design for TinyML Voice-Recognition Application on Resource Frugal Edge Devices'. In: *Applied Sciences 2021, Vol. 11, Page 11073* 11.22 (Nov. 2021), p. 11073. ISSN: 2076-3417. DOI: 10.3390/APP112211073. URL: https://www.mdpi.com/2076-3417/11/22/11073/htm%20https://www.mdpi.com/2076-3417/11/22/11073.

[77] STMicroelectronics. 'STM32F4DISCOVERY - Discovery kit with STM32F407VG MCU * New order code STM32F407G-DISC1 (replaces STM32F4DISCOVERY)'. URL: https://www.st.com/en/evaluation-tools/stm32f4discovery.html.

[78] STMicroelectronics. 'B-L475E-IOT01A - STM32L4 Discovery kit IoT node, low-power wireless, BLE, NFC, SubGHz, Wi-Fi'. URL: https://www.st.com/en/evaluation-tools/b-l475e-iot01a.html.

[79] Eta Compute. 'Eta Compute ECM3532 AI Sensor Product Brief ECM3532 AI Sensor Board: Ultra Low Power Sensor Board for Artificial Intelligence at the Edge'. In: (). URL: http://www.etacompute.com/.

[80] Arduino. 'Nano 33 BLE Sense'. URL: https://docs.arduino.cc/hardware/nano-33-ble-sense.

[81] Sachin Negi and Neeraj Sharma. 'A standalone computing system to classify human foot movements using machine learning techniques for ankle-foot prosthesis control'. In: *https://doi-org.ezproxy2.utwente.nl* (2021). ISSN: 14768259. DOI: 10.1080/10255842.2021.2012656. URL: https://www-tandfonline-com.ezproxy2.utwente.nl/doi/abs/10.1080/10255842.2021.2012656.

[82] OpenMV. 'Quick reference for the openmvcam'. URL: https://docs.openmv.io/openmvcam/quickref.html.

[83] Aditya Jyoti Paul, Puranjay Mohan and Stuti Sehgal. 'Rethinking Generalization in American Sign Language Prediction for Edge Devices with Extremely Low Memory Footprint'. In: *2020 IEEE Recent Advances in Intelligent Computational Systems, RAICS 2020* (Dec. 2020), pp. 147–152. DOI: 10.1109/RAICS51191.2020.9332480.

[84] Sparkfun. 'Himax WE-I Plus EVB Endpoint AI Development Board - DEV-17256'. URL: https://www.sparkfun.com/products/17256.

[85] Silicon Labs. 'IoT Development Kit - Thunderboad Sense 2 - SLTB004A'. URL: https://www.silabs.com/development-tools/thunderboard/thunderboard-sense-two-kit.

[86] Sony. 'Overview - Spresense'. URL: https://developer.sony.com/develop/spresense/.

[87] Syntiant. 'TinyML Board'. URL: https://www.syntiant.com/tinyml.

[88] Arduino. 'Portenta H7'. URL: https://store.arduino.cc/products/portenta-h7.

[89] Raspberry Pi. 'Buy a Raspberry Pi 4 Model B'. URL: https://www.raspberrypi.com/products/raspberry-pi-4-model-b/.

[90] NVIDIA Developer. 'NVIDIA Jetson Nano Developer Kit | NVIDIA Developer'. URL: https://developer.nvidia.com/embedded/jetson-nano-developer-kit. *NVIDIA Developer*.

[91] Bitcraze. 'AI deck 1.1'. URL: https://www.bitcraze.io/products/ai-deck/.

[92] ArduCam. 'Arducam Pico4ML TinyML Dev Kit'. URL: https://www.arducam.com/docs/pico/arducam-pico4mltinymldevkit/.

[93] Arduino. 'Arduino MKR Vidor 4000'. URL: https://store.arduino.cc/products/arduino-mkr-vidor-4000.

[94] Arduino. 'Nicla Sense ME'. URL: https://store.arduino.cc/products/nicla-sense-me.

[95] TI. 'LAUNCHXL-CC1352P Development kit'. URL: https://www.ti.com/tool/LAUNCHXL-CC1352P.

[96] Espressif System. 'ESP-EYE AI Board I'. URL: https://www.espressif.com/en/products/devkits/esp-eye/overview.

[97] GreenWaves Technologies. 'Ultra low power GAP processors'. URL: https://greenwaves-technologies.com/low-power-processor/.

[98] Nordic Semiconductor. 'nRF52840 DK'. URL: https://www.nordicsemi.com/Products/Development-hardware/nrf52840-dk.

[99] Nordic Semiconductor. 'Nordic Thingy:91 Prototyping platform'. URL: https://www.nordicsemi.com/Products/Development-hardware/Nordic-Thingy-91.

[100] NXP Semiconductors. 'FRDM-K64F Platform|Freedom Development Board|Kinetis MCUs'. URL: https://www.nxp.com/design/development-boards/freedom-development-boards/mcu-boards/freedom-development-platform-for-kinetis-k64-k63-and-k24-mcus:FRDM-K64F.

[101] Coral. 'Coral'. URL: https://coral.ai/.

[102] 'Sipeed MAix GO Suit for RISC-V AI+IoT - Seeed Studio'. URL: https://www.seeedstudio.com/Sipeed-MAix-GO-Suit-for-RISC-V-AI-IoT-p-2874.html.

[103] 'OAK-D-Lite — DepthAI Hardware Documentation 1.0.0 documentation'. URL: https://docs.luxonis.com/projects/hardware/en/latest/pages/DM9095.html.

[104] Ahmed I. Awad et al. 'Utilization of mobile edge computing on the Internet of Medical Things: A survey'. In: *ICT Express* (May 2022). ISSN: 2405-9595. DOI: 10.1016/J.ICTE.2022.05.006.

[105] Ramon Sanchez-Iborra and Antonio F. Skarmeta. 'TinyML-Enabled Frugal Smart Objects: Challenges and Opportunities'. In: *IEEE Circuits and Systems Magazine* 20.3 (July 2020), pp. 4–18. ISSN: 15580830. DOI: 10.1109/MCAS.2020.3005467.

[106] Google. 'TensorFlow Lite for Microcontrollers'. URL: https://www.tensorflow.org/lite/microcontrollers.

[107] PyTorch. 'Home | PyTorch'. URL: https://pytorch.org/mobile/home/.

[108] Microsoft. 'The Embedded Learning Library - Embedded Learning Library (ELL)'. URL: https://microsoft.github.io/ELL/.

[109] ARM. 'Arm NN SDK'. URL: https://www.arm.com/products/silicon-ip-cpu/ethos/arm-nn.

[110] CMSIS. 'CMSIS NN Software Library'. URL: https://www.keil.com/pack/doc/CMSIS/NN/html/index.html.

[111] STMicroelectronics. 'AI:X-CUBE-AI documentation - stm32mcu'. URL: https://wiki.stmicroelectronics.cn/stm32mcu/wiki/AI:X-CUBE-AI_documentation.

[112] Fraunhofer-IMS. 'AIfES_for_Arduino: This is the Arduino® compatible port of the AIfES machine learning framework, developed and maintained by Fraunhofer Institute for Microelectronic Circuits and Systems.' URL: https://github.com/Fraunhofer-IMS/AIfES_for_Arduino.

[113] STMicroelectronics. 'Home - NanoEdge™ AI Studio'. URL: https://cartesiam.ai/.

[114] Eloquentarduino. 'micromlgen: Generate C code for microcontrollers from Python's sklearn classifiers'. URL: https://github.com/eloquentarduino/micromlgen.

[115] Nok. 'sklearn-porter: Transpile trained scikit-learn estimators to C, Java, JavaScript and others.' URL: https://github.com/nok/sklearn-porter.

[116] BayesWitnesses. 'm2cgen: Transform ML models into a native code (Java, C, Python, Go, JavaScript, Visual Basic, C#, R, PowerShell, PHP, Dart, Haskell, Ruby, F#, Rust) with zero dependencies'. URL: https://github.com/BayesWitnesses/m2cgen.

[117] Nok. 'weka-porter: Transpile trained decision trees from Weka to C, Java or JavaScript.' URL: https://github.com/nok/weka-porter.

[118] Lucastsutsui. 'EmbML: A tool to support using classification models in low-power and microcontroller-based embedded systems.' URL: https://github.com/lucastsutsui/EmbML.

[119] Emlearn. 'emlearn: Machine Learning inference engine for Microcontrollers and Embedded devices'. URL: https://github.com/emlearn/emlearn.

[120] UTensor. 'uTensor: TinyML AI inference library'. URL: https://github.com/uTensor/uTensor.

[121] Eloquentarduino. 'tinymlgen: Generate C code for microcontrollers from Tensorflow models'. URL: https://github.com/eloquentarduino/tinymlgen.

[122] EEESlab. 'CMix-NN: CMix-NN: Mixed Low-Precision CNN Library for Memory-Constrained Edge Devices'. URL: https://github.com/EEESlab/CMix-NN.

[123] Pulp-platform. 'fann-on-mcu'. URL: https://github.com/pulp-platform/fann-on-mcu.

[124] Neuton. 'Neuton AI TinyML'. URL: https://neuton.ai/.

[125] Edge Impulse. 'Adding sight to your sensors'. URL: https://docs.edgeimpulse.com/docs/tutorials/image-classification%20https://docs.edgeimpulse.com/docs/image-classification.

[126] TensorFlow. 'Building Machine Learning models with Edge Impulse - YouTube'. URL: https://www.youtube.com/watch?v=gw1E5JZTim0.

[127] Oduor_c. 'Tflite to be used with Google Coral'. URL: https://forum.edgeimpulse.com/t/tflite-to-be-used-with-google-coral/3891/5.

[128] A Krizhevsky, V Nair and G Hinton. 'CIFAR-10 and CIFAR-100 datasets'. URL: https://www.cs.toronto.edu/~kriz/cifar.html.

[129] Stanford Vision Lab, Stanford University and Princeton University. 'ImageNet'. URL: https://www.image-net.org/index.php.

[130] Microsoft. 'COCO - Common Objects in Context'. URL: https://cocodataset.org/#explore%20https://cocodataset.org/#home.

[131] Walter J. Scheirer et al. 'Toward open set recognition'. In: *IEEE Transactions on Pattern Analysis and Machine Intelligence* 35.7 (2013), pp. 1757–1772. ISSN: 01628828. DOI: 10.1109/TPAMI.2012.256.

[132] Mark Sandler et al. 'MobileNetV2: Inverted Residuals and Linear Bottlenecks'. In: (Jan. 2018). DOI: 10.48550/arxiv.1801.04381. URL: https://arxiv.org/abs/1801.04381v4.

[133] Mingxing Tan and Quoc V. Le. 'EfficientNet: Rethinking Model Scaling for Convolutional Neural Networks'. In: *36th International Conference on Machine Learning, ICML 2019* 2019-June (May 2019), pp. 10691–10700. DOI: 10.48550/arxiv.1905.11946. URL: https://arxiv.org/abs/1905.11946v5.

[134] Mingxing Tan and Quoc V. Le. 'EfficientNetV2: Smaller Models and Faster Training'. In: (Apr. 2021). DOI: 10.48550/arxiv.2104.00298. URL: https://arxiv.org/abs/2104.00298v3.

[135] Christian Szegedy et al. 'Inception-v4, Inception-ResNet and the Impact of Residual Connections on Learning'. In: ().

[136] Google Coral. 'TensorFlow models on the Edge TPU'. URL: https://coral.ai/docs/edgetpu/models-intro/.

[137] Google. 'Edge TPU performance benchmarks'. URL: https://coral.ai/docs/edgetpu/benchmarks/%20https://coral.ai/docs/edgetpu/benchmarks. *Coral*.

[138] Gao Huang et al. 'Densely Connected Convolutional Networks'. In: *Proceedings - 30th IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2017* 2017-January (Aug. 2016), pp. 2261–2269. DOI: 10.48550/arxiv.1608.06993. URL: https://arxiv.org/abs/1608.06993v5.

[139] Andrew G. Howard et al. 'MobileNets: Efficient Convolutional Neural Networks for Mobile Vision Applications'. In: (Apr. 2017). DOI: 10.48550/arxiv.1704.04861. URL: https://arxiv.org/abs/1704.04861v1.

[140] Kaiming He et al. 'Deep Residual Learning for Image Recognition'. In: *Proceedings of the IEEE Computer Society Conference on Computer Vision and Pattern Recognition* 2016-December (Dec. 2015), pp. 770–778. ISSN: 10636919. DOI: 10.48550/arxiv.1512.03385. URL: https://arxiv.org/abs/1512.03385v1.

[141] Kaiming He et al. 'Identity Mappings in Deep Residual Networks'. In: (). URL: https://github.com/KaimingHe/.

[142] Suyog Gupta and Mingxing Tan. 'Google AI Blog: EfficientNet-EdgeTPU: Creating Accelerator-Optimized Neural Networks with AutoML'. URL: https://ai.googleblog.com/2019/08/efficientnet-edgetpu-creating.html.

[143] Christian Szegedy et al. 'Going deeper with convolutions'. In: ().

[144] Andrew Howard et al. 'Searching for MobileNetV3'. In: (May 2019). DOI: 10.48550/arxiv.1905.02244. URL: https://arxiv.org/abs/1905.02244v5.

[145] Ji Lin et al. 'MCUNetV2: Memory-Efficient Patch-based Inference for Tiny Deep Learning'. In: (). URL: https://mcunet.mit.edu.

[146] Alexander Wong, Mahmoud Famouri and Mohammad Javad Shafiee. 'AttendNets: Tiny Deep Image Recognition Neural Networks for the Edge via Visual Attention Condensers'. In: ().

[147] Oindrila Saha et al. 'RNNPool: Efficient Non-linear Pooling for RAM Constrained Inference'. In: (). URL: https://github.com/Microsoft/EdgeML..

[148] KS30. 'Accuracy vs F1 score | Data Science and Machine Learning | Kaggle'. URL: https://www.kaggle.com/questions-and-answers/178137.

[149] Avijit Hazra. 'Using the confidence interval confidently'. In: *Journal of Thoracic Disease* 9.10 (Oct. 2017), pp. 4125–4130. ISSN: 20776624. DOI: 10.21037/jtd.2017.09.14. URL: https://www.researchgate.net/publication/320742650_Using_the_confidence_interval_confidently.

[150] University of Twente. 'Research support: Jupyter | JupyterLab | Jupiter | Cloud computing | Service Portal'. URL: https://www.utwente.nl/en/service-portal/research-support/research-support-topics/it-facilities-for-research/jupyterlab.

[151] WandB. 'Weights & Biases - Developer tools for ML'. URL: https://wandb.ai/site. *Wandb*.

[152] Lutz Roeder. 'Netron'. URL: https://netron.app/.

[153] OpenCV. 'Home - OpenCV'. URL: https://opencv.org/.

[154] NVIDIA. 'NVIDIA T4 TENSOR CORE GPU SPECIFICATIONS GPU Architecture NVIDIA Turing NVIDIA Turing Tensor Cores 320 NVIDIA CUDA ® Cores 2,560'. In: (2018), pp. 8–9. URL: www.nvidia.com/T4.

[155] Consult Red. 'Feature Processor Arm GreenWaves'. In: (). URL: https://consult.red/discover-red/.

[156] Ubuntu. 'WSL | Ubuntu'. URL: https://ubuntu.com/wsl.

[157] Google. 'Edge TPU Compiler | Coral'. URL: https://coral.ai/docs/edgetpu/compiler/%20https://coral.ai/docs/edgetpu/compiler/#co-compiling-multiple-models.

[158] Google. 'Camera | Coral'. URL: https://coral.ai/products/camera/%20https://coral.ai/products/camera/#tech-specs.

[159] e-con Systems. 'Cameras for Google Coral Dev Board'. URL: https://www.e-consystems.com/cameras-for-google-coral.asp.

[160] Logitech. 'Logitech C920 PRO HD Webcam, 1080p Video with Stereo Audio'. URL: https://www.logitech.com/en-gb/products/webcams/c920-pro-hd-webcam.960-001055.html.

[161] Google. 'TensorFlow models on the Edge TPU'. URL: https://coral.ai/docs/edgetpu/models-intro/#quantization%20https://coral.ai/docs/edgetpu/models-intro/#supported-operations. *Coral*.

[162] The Things Network. 'The Things Network'. URL: https://www.thethingsnetwork.org/docs/lorawan/limitations/.

[163] Taoufik Bouguera et al. 'Energy consumption model for sensor nodes based on LoRa and LoRaWAN Energy consumption model for sensor nodes based Energy Consumption Model for Sensor Nodes Based on LoRa and LoRaWAN'. In: *Sensors* 18.7 (2018). DOI: 10.3390/s18072104{\"{i}}. URL: https://hal.science/hal-01828769.

[164] Li Peng. 'Comparing 4G and 5G downlink energy consumption'. In: (2022), pp. 1–6. DOI: 10.36227/techrxiv.21269118.v1.

[165] Emil Bjornson and Erik G Larsson. 'How Energy-Efficient Can a Wireless Communication System Become?' In: *Conference Record - Asilomar Conference on Signals, Systems and Computers*. Vol. 2018-Octob. 2019, pp. 1252–1256. ISBN: 9781538692189. DOI: 10.1109/ACSSC.2018.8645227.

[166] Ming Yan et al. 'Modeling the total energy consumption of mobile network services and applications'. In: *Energies* 12.1 (Jan. 2019). ISSN: 19961073. DOI: 10.3390/EN12010184. URL: https://www.researchgate.net/publication/330201584_Modeling_the_Total_Energy_Consumption_of_Mobile_Network_Services_and_Applications.

[167] Nhu Ho. 'LTE-M vs NB-IoT'. URL: https://www.emnify.com/blog/lte-m-nb-iot. *EMnify Blog*.

[168] 1NCE Data Broker. 'Enhancing Battery Life on NB-IoT and LTE-M'. URL: https://1nce.com/en-eu/resources/news-insights/blog/1nce-data-broker.

[169] Harald Naumann. '(26) SIGFOX versus NB-IoT - power estimation'. URL: https://www.linkedin.com/pulse/sigfox-versus-nb-iot-power-consumption-harald-naumann/.

[170] Harald Naumann. 'NB-IoT versus SIGFOX, LoRaWAN, and Weightless – power / energy the inconvenient truth'. URL: https://www.gsm-modem.de/M2M/iot-university/nb-iot-power-consumption/.

[171] Sigfox. 'Sigfox Technical Overview'. In: *Onsemi* 1.January (2017), p. 26. URL: https://www.disk91.com/wp-content/uploads/2017/05/4967675830228422064.pdf.

[172] Gunnar P Noordbruis. 'Energy efficient WLAN using WiFi standards b/a/g/n/ac on an Archer C7 AC1750 access point'. In: (2020).

[173] William Antonelli and John Lynch. 'What's a Good Internet Speed? How to Upgrade Your Internet'. URL: https://www.businessinsider.com/guides/tech/what-is-a-good-internet-speed?international=true&r=US&IR=T.

[174] Hamed Valizadegan, Quang Nguyen and Milos Hauskrecht. 'Learning Classification Models from Multiple Experts'. In: *Journal of biomedical informatics* 46.6 (Dec. 2013), p. 1125. ISSN: 15320464. DOI: 10.1016/J.JBI.2013.08.007. URL: /pmc/articles/PMC3922063/%20/pmc/articles/PMC3922063/?report=abstract%20https://www.ncbi.nlm.nih.gov/pmc/articles/PMC3922063/.

[175] Arish Sateesan et al. 'A Survey of Algorithmic and Hardware Optimization Techniques for Vision Convolutional Neural Networks on FPGAs'. In: *Neural Processing Letters* 53.3 (June 2021), pp. 2331–2377. ISSN: 1573773X. DOI: 10.1007/S11063-021-10458-1/TABLES/10. URL: https://link.springer.com/article/10.1007/s11063-021-10458-1.

[176] Keysight Technologies Inc. '34465A Digital Multimeter, 6½ Digit, Performance Truevolt DMM'. URL: https://www.keysight.com/us/en/product/34461A/digital-multimeter-6-5-digit-truevolt-dmm.html%20http://www.keysight.com/en/pd-2519397-pn-34465A/digital-multimeter-6-digit-performance-truevolt-dmm?cc=DE&lc=ger.

[177] Nikiforos Pittaras et al. 'Comparison of fine-tuning and extension strategies for deep convolutional neural networks'. In: *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)* 10132 LNCS (2017), pp. 102–114. ISSN: 16113349. DOI: 10.1007/978-3-319-51811-4{_}9.

[178] Saulo Baretto. 'What is a Monitor in Computer Science? | Baeldung on Computer Science'. URL: https://www.baeldung.com/cs/fine-tuning-nn%20https://www.baeldung.com/cs/monitor.

[179] Aston Zhang et al. 'Dive into Deep Learning'. In: *Journal of the American College of Radiology* 17.5 (June 2021), pp. 637–638. ISSN: 1558349X. DOI: 10.1016/j.jacr.2020.02.005. URL: https://arxiv.org/abs/2106.11342v4.

[180] Google Coral. 'Retrain a classification model on-device with weight imprinting | Coral'. URL: https://coral.ai/docs/edgetpu/retrain-classification-ondevice/%20https://coral.ai/docs/edgetpu/retrain-classification-ondevice/#overview.

[181] Google Coral. 'Retrain a classification model on-device with backpropagation | Coral'. URL: https://coral.ai/docs/edgetpu/retrain-classification-ondevice-backprop/.

12

APPENDICES

A RESULTS EXPERIMENT 1

A.1 *Individual Distribution Accuracies*

Table 20: Mean accuracy and 95% Confidence Interval of algorithms tested on all distribution versus the number of evaluation images per class. The distributions on the left were the distributions used as the evaluation set.

Algorithm	1	2	5	10	15	25	50	75	100
MobileNetV2									
Cityscape	0.51 ± 0.035	0.515 ± 0.077	0.48 ± 0.043	0.477 ± 0.027	0.471 ± 0.016	0.48 ± 0.014	0.485 ± 0.01	0.481 ± 0.005	0.482 ± 0.0
Forest	0.61 ± 0.122	0.655 ± 0.052	0.672 ± 0.029	0.678 ± 0.032	0.674 ± 0.023	0.654 ± 0.019	0.651 ± 0.007	0.655 ± 0.006	0.655 ± 0.
Office	0.64 ± 0.078	0.56 ± 0.05	0.572 ± 0.033	0.583 ± 0.028	0.591 ± 0.019	0.591 ± 0.009	0.577 ± 0.006	0.587 ± 0.003	0.586 ± 0.0
Park	0.4 ± 0.092	0.43 ± 0.066	0.392 ± 0.02	0.423 ± 0.025	0.415 ± 0.023	0.431 ± 0.013	0.413 ± 0.011	0.42 ± 0.005	0.419 ± 0.0
Pub	0.67 ± 0.083	0.695 ± 0.037	0.684 ± 0.031	0.68 ± 0.021	0.689 ± 0.011	0.675 ± 0.011	0.688 ± 0.008	0.688 ± 0.004	0.688 ± 0.0
Uniform	0.94 ± 0.043	0.85 ± 0.074	0.882 ± 0.03	0.895 ± 0.019	0.895 ± 0.011	0.894 ± 0.005	0.891 ± 0.006	0.89 ± 0.004	0.891 ± 0.0
EfficientNetB0									
Cityscape	0.62 ± 0.076	0.635 ± 0.039	0.65 ± 0.034	0.636 ± 0.032	0.629 ± 0.016	0.629 ± 0.013	0.63 ± 0.006	0.632 ± 0.004	0.63 ± 0.0
Forest	0.82 ± 0.049	0.845 ± 0.056	0.832 ± 0.031	0.857 ± 0.018	0.813 ± 0.011	0.812 ± 0.012	0.808 ± 0.008	0.815 ± 0.004	0.815 ± 0.0
Office	0.69 ± 0.068	0.655 ± 0.071	0.672 ± 0.043	0.702 ± 0.02	0.713 ± 0.014	0.693 ± 0.007	0.693 ± 0.009	0.698 ± 0.003	0.696 ± 0.0
Park	0.58 ± 0.116	0.565 ± 0.076	0.556 ± 0.019	0.59 ± 0.019	0.589 ± 0.016	0.592 ± 0.011	0.587 ± 0.005	0.58 ± 0.005	0.583 ± 0.0
Pub	0.8 ± 0.065	0.825 ± 0.039	0.82 ± 0.033	0.815 ± 0.012	0.801 ± 0.022	0.817 ± 0.013	0.807 ± 0.004	0.805 ± 0.005	0.808 ± 0.0
Uniform	0.91 ± 0.02	0.915 ± 0.021	0.944 ± 0.023	0.949 ± 0.009	0.947 ± 0.01	0.941 ± 0.005	0.938 ± 0.004	0.939 ± 0.004	0.939 ± 0.0
EfficientNetV2B0									
Cityscape	0.77 ± 0.066	0.74 ± 0.058	0.736 ± 0.027	0.741 ± 0.024	0.739 ± 0.022	0.745 ± 0.012	0.731 ± 0.007	0.733 ± 0.003	0.732 ± 0.0
Forest	0.87 ± 0.051	0.87 ± 0.03	0.864 ± 0.018	0.859 ± 0.015	0.852 ± 0.01	0.84 ± 0.013	0.857 ± 0.008	0.851 ± 0.003	0.853 ± 0.0
Office	0.75 ± 0.073	0.785 ± 0.07	0.756 ± 0.031	0.781 ± 0.024	0.767 ± 0.012	0.767 ± 0.011	0.761 ± 0.01	0.764 ± 0.004	0.766 ± 0.0
Park	0.58 ± 0.087	0.545 ± 0.049	0.558 ± 0.045	0.561 ± 0.023	0.576 ± 0.016	0.563 ± 0.017	0.575 ± 0.009	0.571 ± 0.006	0.571 ± 0.0
Pub	0.83 ± 0.072	0.86 ± 0.046	0.866 ± 0.023	0.85 ± 0.012	0.855 ± 0.014	0.855 ± 0.01	0.847 ± 0.008	0.852 ± 0.004	0.852 ± 0.0
Uniform	0.92 ± 0.049	0.92 ± 0.042	0.952 ± 0.019	0.952 ± 0.009	0.962 ± 0.007	0.961 ± 0.006	0.955 ± 0.004	0.956 ± 0.002	0.954 ± 0.0
EfficientNetV2S									
Cityscape	0.78 ± 0.109	0.805 ± 0.056	0.804 ± 0.024	0.799 ± 0.016	0.821 ± 0.013	0.822 ± 0.017	0.819 ± 0.008	0.825 ± 0.004	0.822 ± 0.0
Forest	0.89 ± 0.054	0.94 ± 0.035	0.934 ± 0.019	0.921 ± 0.027	0.934 ± 0.013	0.924 ± 0.007	0.931 ± 0.006	0.93 ± 0.003	0.931 ± 0.0
Office	0.78 ± 0.082	0.81 ± 0.028	0.806 ± 0.045	0.803 ± 0.024	0.79 ± 0.012	0.797 ± 0.014	0.784 ± 0.007	0.787 ± 0.005	0.789 ± 0.0
Park	0.6 ± 0.072	0.675 ± 0.069	0.66 ± 0.041	0.686 ± 0.02	0.662 ± 0.022	0.684 ± 0.018	0.679 ± 0.004	0.672 ± 0.004	0.677 ± 0.0
Pub	0.92 ± 0.049	0.875 ± 0.044	0.91 ± 0.026	0.893 ± 0.017	0.899 ± 0.01	0.902 ± 0.011	0.9 ± 0.007	0.904 ± 0.003	0.9 ± 0.0
Uniform	0.99 ± 0.02	0.96 ± 0.02	0.974 ± 0.013	0.964 ± 0.006	0.959 ± 0.006	0.958 ± 0.002	0.964 ± 0.002	0.962 ± 0.001	0.963 ± 0.0
InceptionResNetV2									
Cityscape	0.65 ± 0.084	0.575 ± 0.064	0.612 ± 0.029	0.604 ± 0.026	0.611 ± 0.017	0.612 ± 0.009	0.609 ± 0.007	0.607 ± 0.005	0.608 ± 0.0
Forest	0.83 ± 0.083	0.795 ± 0.049	0.824 ± 0.033	0.791 ± 0.03	0.807 ± 0.015	0.807 ± 0.008	0.807 ± 0.008	0.807 ± 0.004	0.808 ± 0.0
Office	0.65 ± 0.098	0.675 ± 0.049	0.662 ± 0.042	0.665 ± 0.025	0.641 ± 0.014	0.636 ± 0.008	0.644 ± 0.01	0.646 ± 0.004	0.648 ± 0.0
Park	0.51 ± 0.094	0.53 ± 0.082	0.512 ± 0.044	0.51 ± 0.033	0.491 ± 0.025	0.482 ± 0.019	0.492 ± 0.01	0.495 ± 0.005	0.495 ± 0.0
Pub	0.76 ± 0.089	0.87 ± 0.026	0.852 ± 0.029	0.815 ± 0.018	0.821 ± 0.013	0.807 ± 0.013	0.813 ± 0.006	0.817 ± 0.004	0.818 ± 0.0
Uniform	0.93 ± 0.083	0.945 ± 0.023	0.952 ± 0.012	0.958 ± 0.01	0.958 ± 0.008	0.955 ± 0.007	0.956 ± 0.004	0.957 ± 0.002	0.956 ± 0.0

A.2 *Plots*

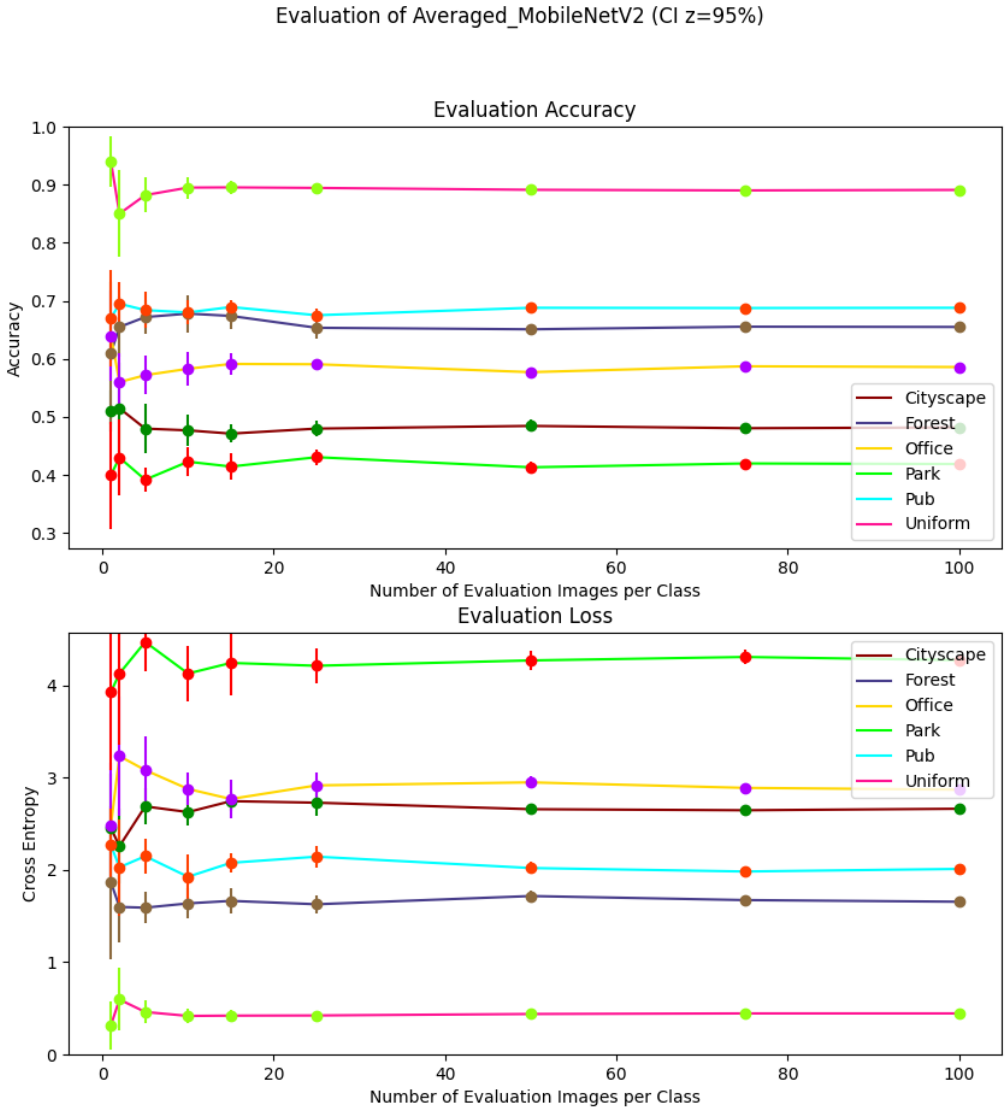


Figure 43: Average accuracy and loss with Confidence Interval of MobileNetV2.

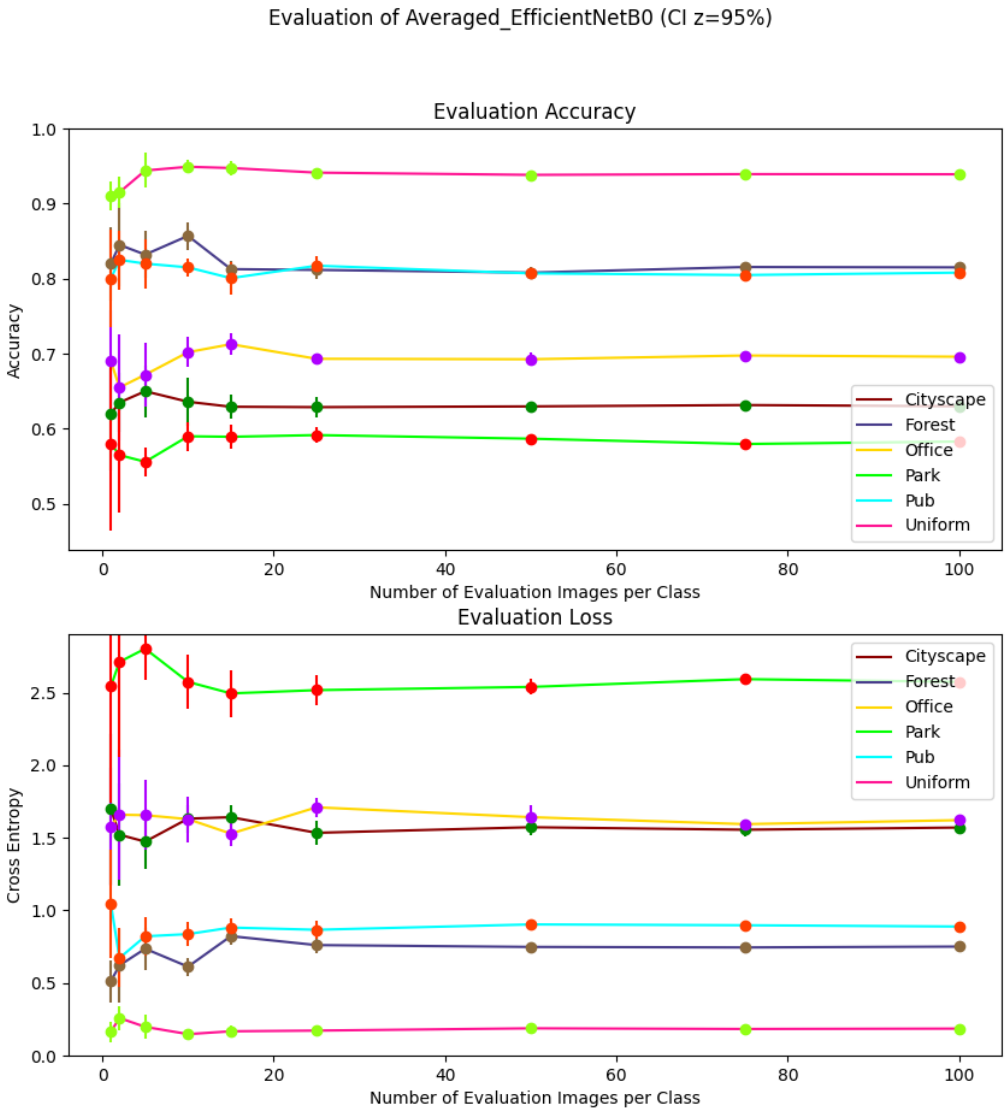


Figure 44: Average accuracy and loss with Confidence Interval of EfficientNetB0.

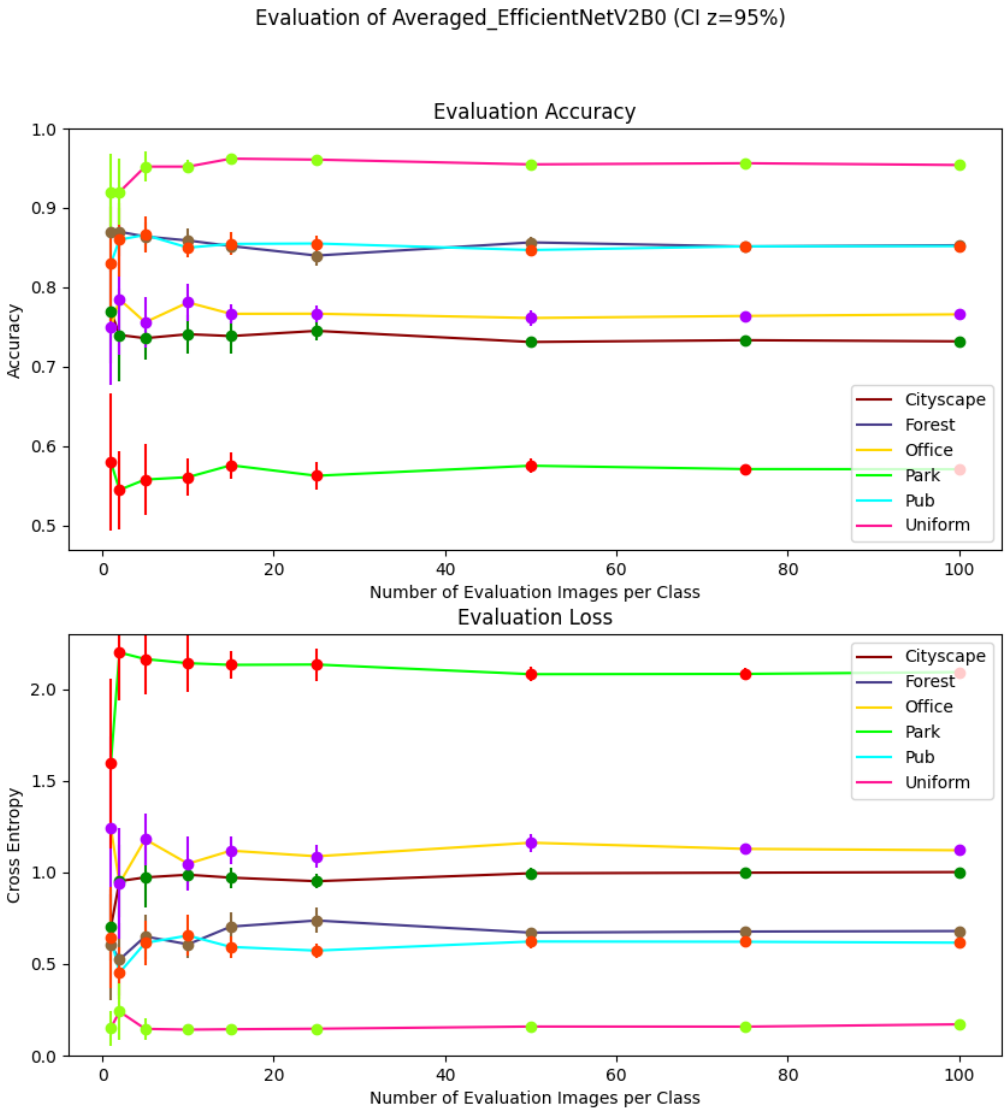


Figure 45: Average accuracy and loss with Confidence Interval of EffiecientNetV2B0.

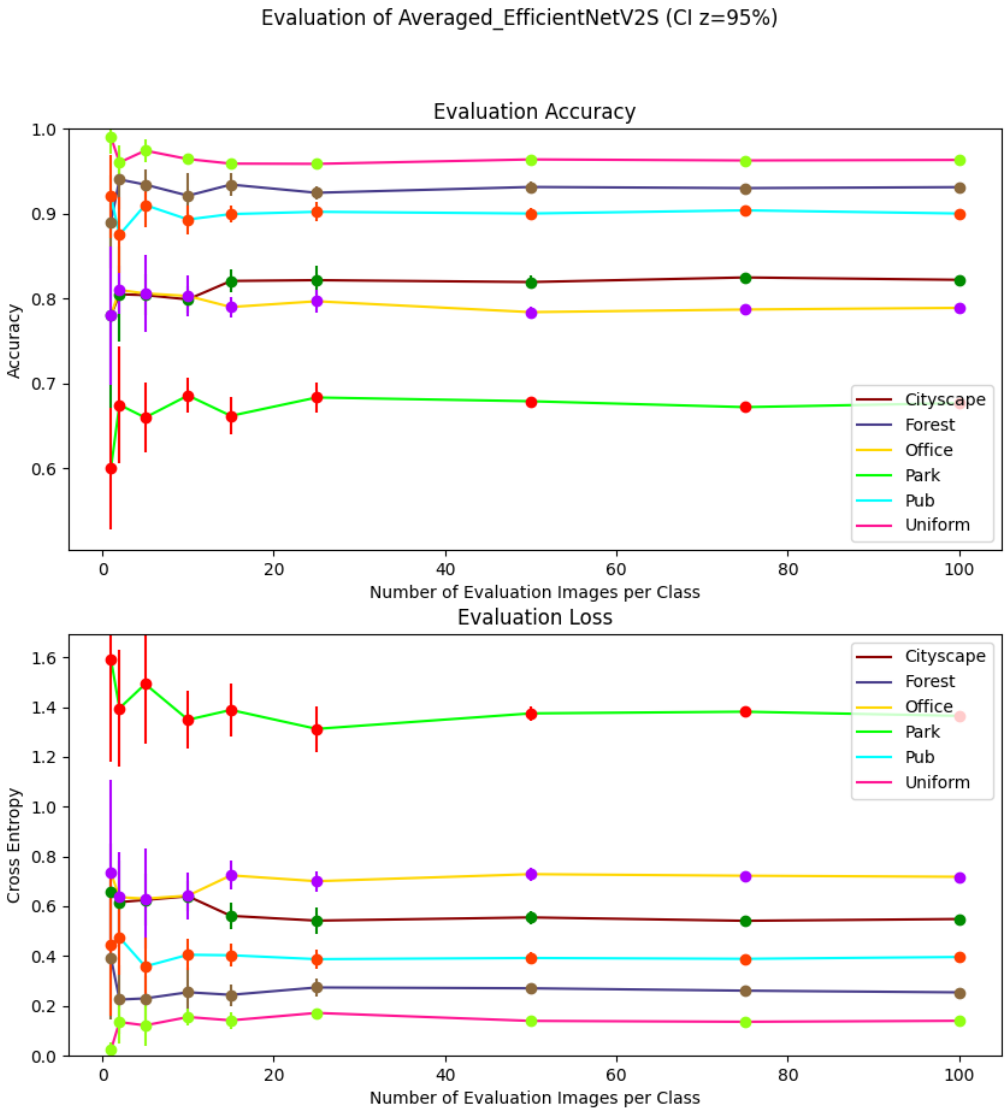


Figure 46: Average accuracy and loss with Confidence Interval of EfficientNetV2S.

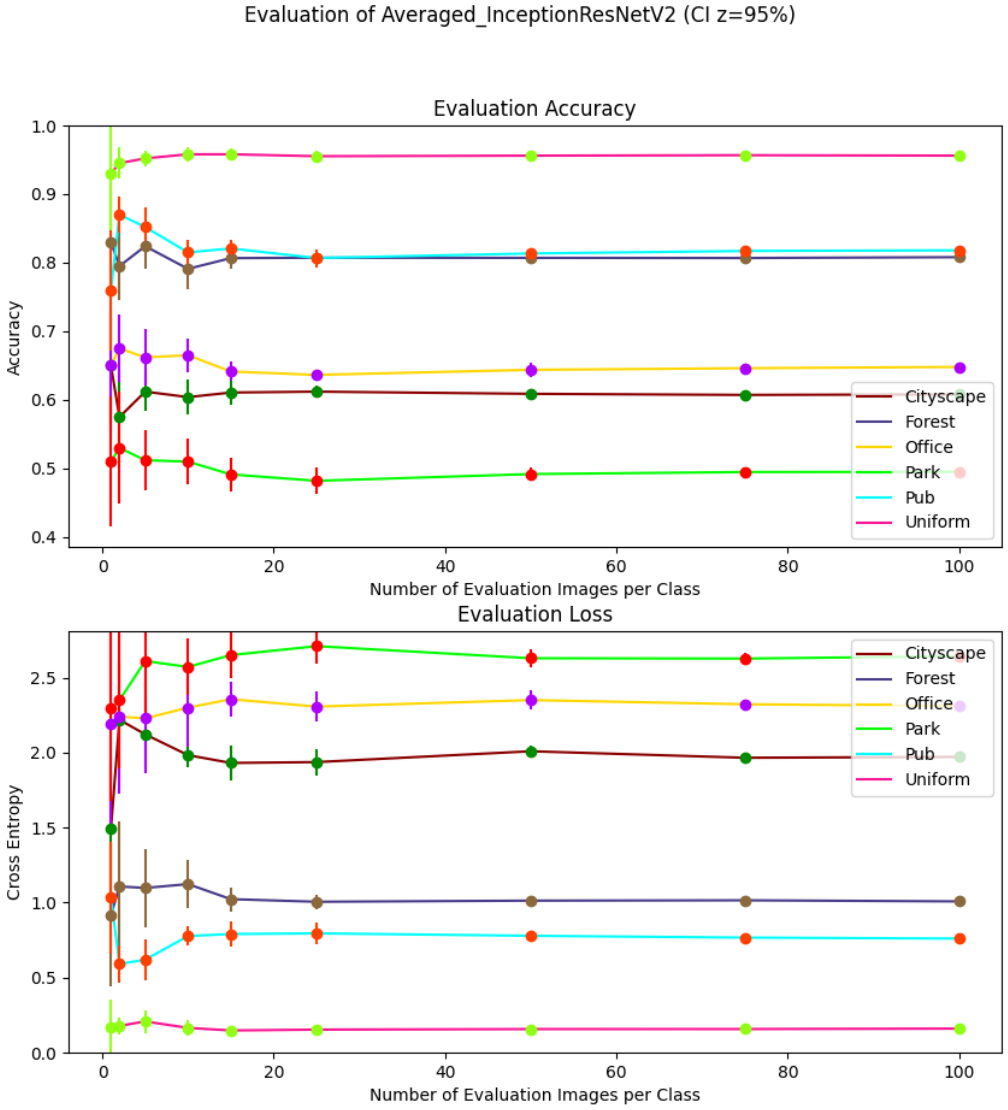


Figure 47: Average accuracy and loss with Confidence Interval of InceptionResNetV2.

B RESULTS EXPERIMENT 3

B.1 *Classification Performances*

Table 21: The performance of the quantised and full models. For every model, the accuracy of the summed confusion matrixes is given, as well as the F1-score, precision, recall and time.

Algorithm	Full					Quantised				
	Accuracy	F1-score	Precision	Recall	Time (ms/image)	Accuracy	F1-score	Precision	Recall	Time (ms/image)
MobileNetV2	0.6202	0.6194	0.6202	0.6459	5.859	0.5997	0.5986	0.6265	0.5997	**29.917**
EfficientNetB0	0.7452	0.7423	0.7590	0.7452	4.505	0.7417	0.7378	0.7497	0.7417	58.396
EfficientNetV2B0	0.7882	0.7875	0.8077	0.7882	**3.934**	0.7705	0.7702	0.7961	0.7705	47.070
EfficientNetV2S	**0.8470**	**0.8479**	**0.8523**	**0.8470**	12.831	**0.8525**	**0.8535**	**0.8584**	**0.8525**	390.206
InceptionResNetV2	0.7303	0.7371	0.7596	0.7303	10.764	0.7275	0.7337	0.7555	0.7275	347.916

Table 22: Here the quantised models are shown running on CPU and TPU. For every model, the accuracy of the summed confusion matrixes is given, as well as the overall F1-score, precision and recall.

Algorithm	Quantised running on CPU				Quantised running on TPU			
	Acc	F1-score	Precision	Recall	Acc	F1-score	Precision	Recall
MobileNetV2	0.4312	0.4398	0.4993	0.4312	0.4313	0.4396	0.4976	0.4313
EfficientNetB0	0.6110	0.6152	0.6382	0.6110	0.6048	0.6091	0.6345	0.6048
EfficientNetV2B0	0.6808	0.6901	0.7205	0.6808	**0.6778**	**0.6883**	**0.7227**	**0.6778**
EfficientNetV2S	**0.7778**	**0.7814**	**0.7975**	**0.7778**	N.A.	N.A.	N.A.	N.A.
InceptionResNetV2	0.6128	0.6257	0.6614	0.6128	0.6142	0.6272	0.6629	0.6142

B.2 *Summed Confusion Matrices*

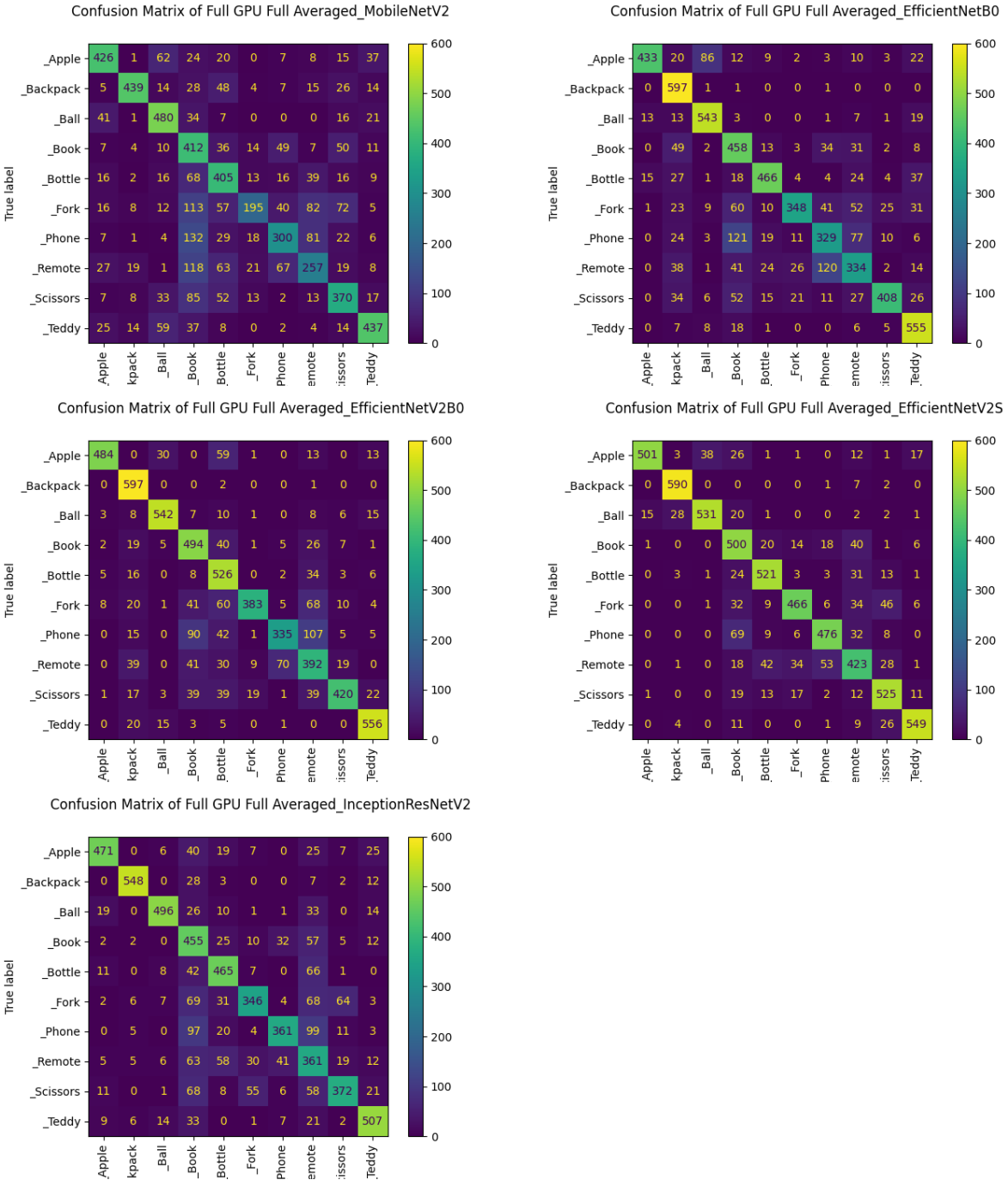


Figure 48: Summed confusion matrixes of the full models ran on Jupyter Lab in experiment 3.

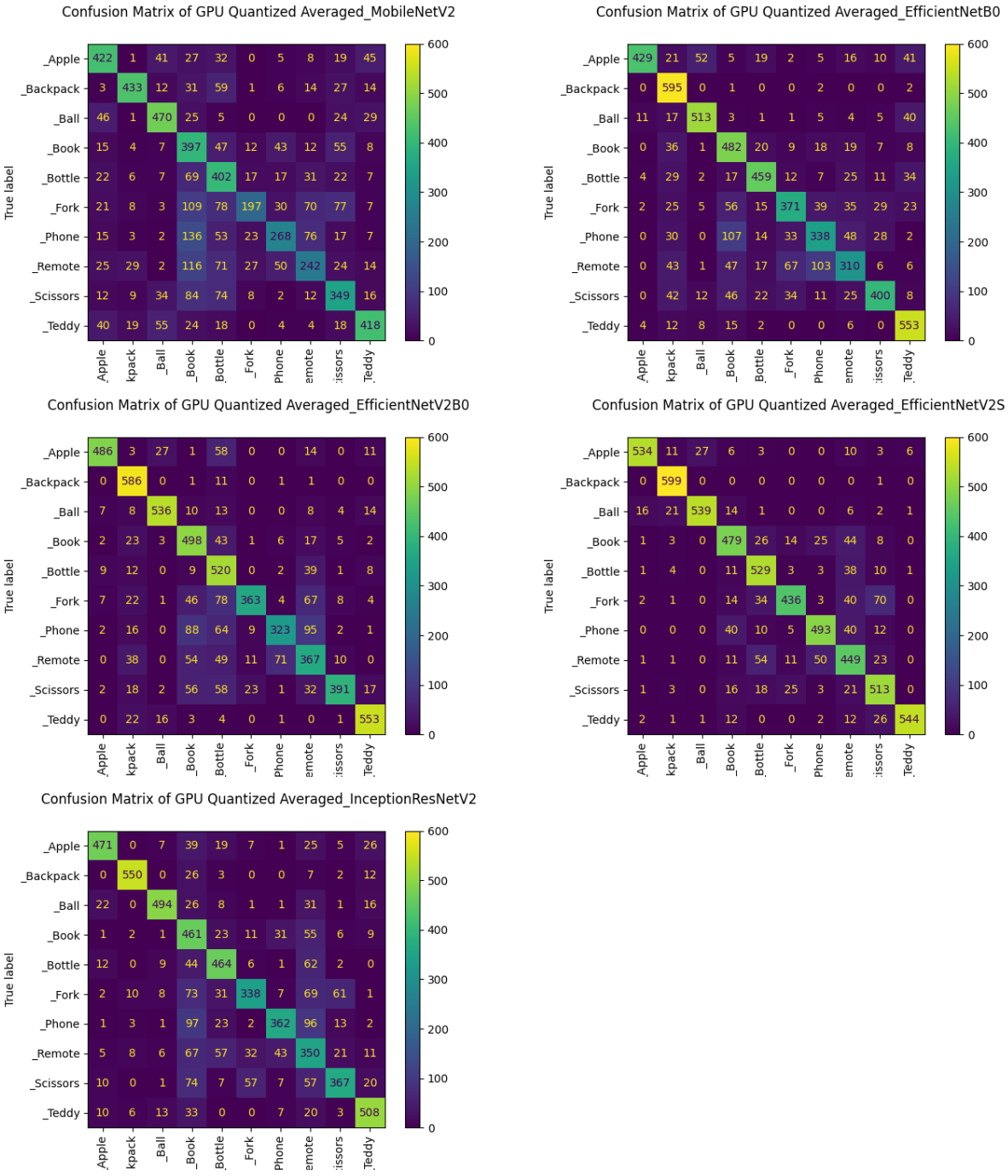


Figure 49: Summed confusion matrixes of the quantised models ran on Jupyter Lab in experiment 3.

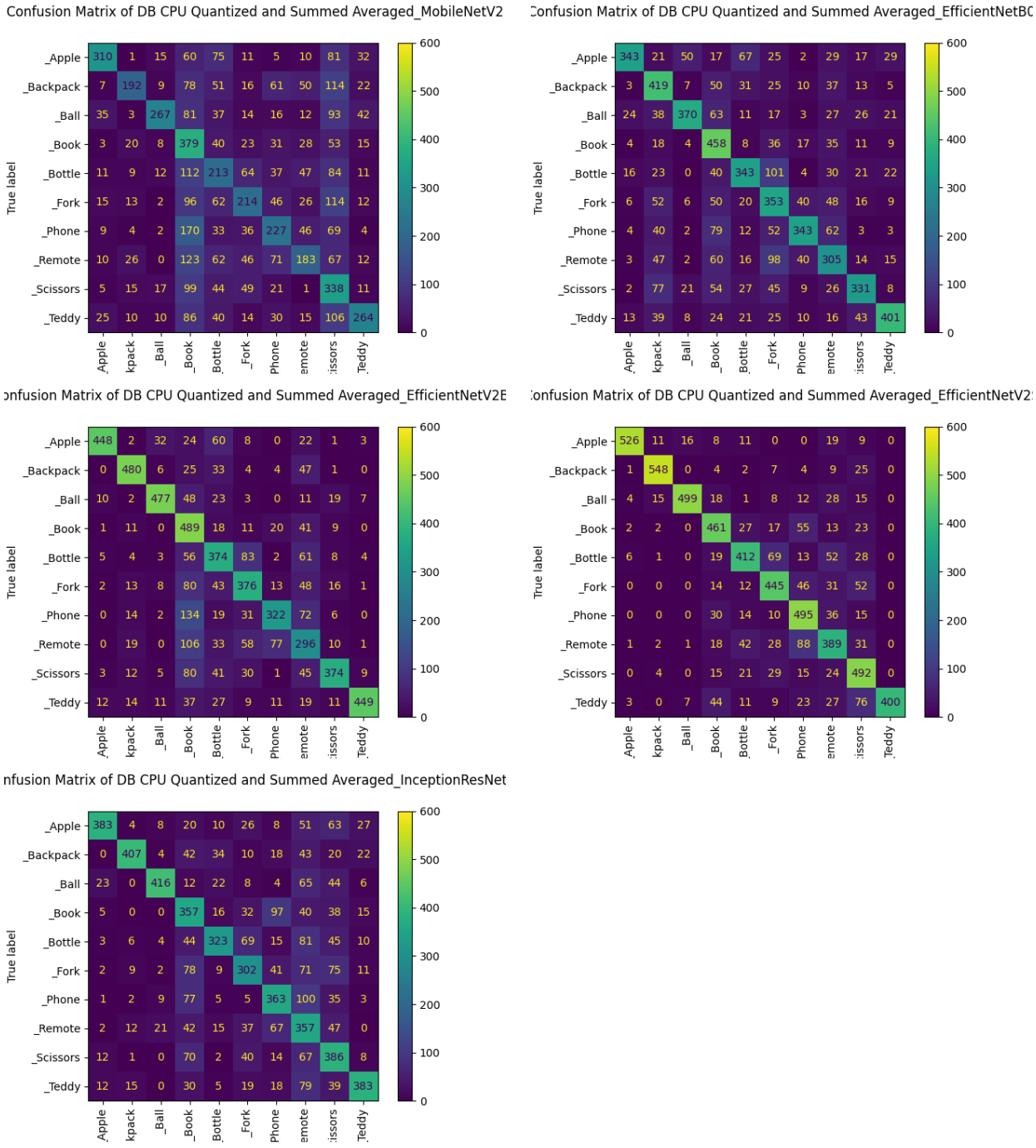


Figure 50: Summed confusion matrixes of the quantised models ran on the edge CPU in experiment 3.

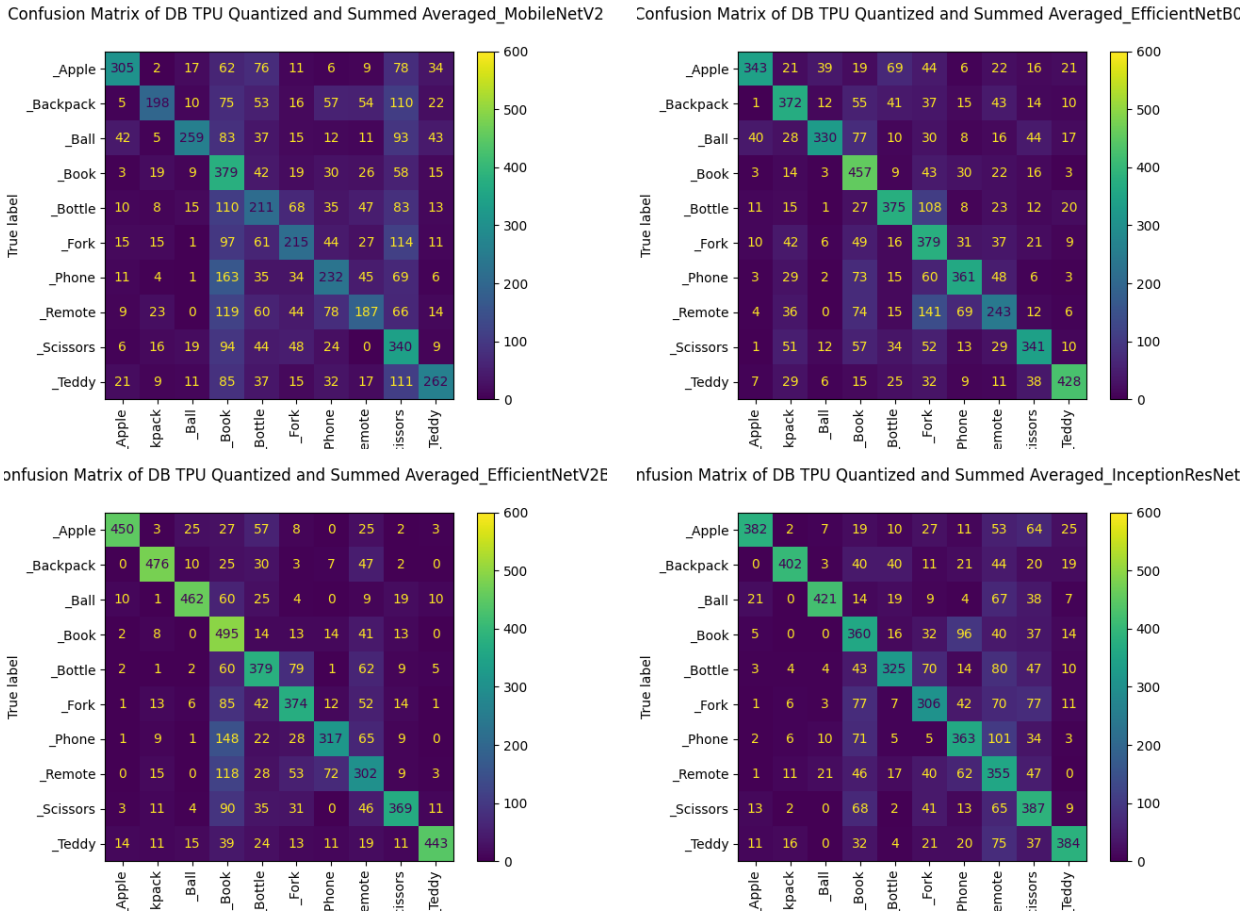


Figure 51: Summed confusion matrixes of the quantised models ran on the edge TPU in experiment 3.

Table 23: The accuracy for the fine-tuned models in experiment 4.

Algorithm	1	5	10	15	20	30	50	80	In Distribution trained 80
MobileNetV2									0.9248
GPU LLSLR	0.6397	0.6747	0.6900	0.7098	0.7215	0.7488	0.7915	0.8408	
GPU ALSLR	0.6457	0.6770	0.6997	0.7422	0.7533	0.7818	0.8253	0.8427	
GPU LLRT	0.9133	0.9250	0.9307	0.9420	0.9520	0.9617	0.9662	0.9780	
GPU LLSLR on Edge	0.4530	0.4700	0.4765	0.4880	0.4965	0.5015	0.5155	0.5332	
GPU ALSLR on Edge	0.4497	0.4717	0.4813	0.4947	0.5118	0.5343	0.5542	0.5620	
GPU LLRT on Edge	0.5323	0.5345	0.5473	0.5585	0.5462	0.5518	0.5610	0.5473	
EfficientNetB0									0.9807
GPU LLSLR	0.7698	0.8068	0.8383	0.8597	0.8760	0.9043	0.9345	0.9620	
GPU ALSLR	0.7632	0.7807	0.8015	0.8185	0.8475	0.8858	0.9338	0.9792	
GPU LLRT	0.9473	0.9472	0.9570	0.9657	0.9698	0.9752	0.9743	0.9843	
GPU LLSLR on Edge	0.6113	0.6268	0.6418	0.6475	0.6557	0.6587	0.6658	0.6780	
GPU ALSLR on Edge	0.6167	0.6120	0.6148	0.6310	0.6305	0.6540	0.6828	0.6827	
GPU LLRT on Edge	0.6220	0.6337	0.6480	0.6458	0.6542	0.6600	0.6652	0.6592	
EfficientNetV2B0									0.9815
GPU LLSLR	0.8028	0.8298	0.8590	0.8790	0.8930	0.9138	0.9367	0.9560	
GPU ALSLR	0.7850	0.8023	0.8190	0.8783	0.9003	0.9437	0.9747	0.9918	
GPU LLRT	0.9473	0.9530	0.9598	0.9678	0.9683	0.9728	0.9808	0.9858	
GPU LLSLR on Edge	0.6860	0.6970	0.7088	0.7183	0.7228	0.7277	0.7425	0.7565	
GPU ALSLR on Edge	0.6600	0.6418	0.5568	0.5750	0.5680	0.6142	0.6515	0.6932	
GPU LLRT on Edge	0.7060	0.6963	0.7043	0.7137	0.7232	0.7223	0.7432	0.7355	

Table 24: Performance of the full models with 80 images per class of In-Distribution data with 5-fold cross-validation.

Algorithm	Full model with In Distribution Data				
	Accuracy	F1-score	Precision	Recall	Time (ms/image)
MobileNetV2	0.9248	0.9249	0.9255	0.9248	12.005
EfficientNetB0	0.9807	0.9807	0.9808	0.9807	9.493
EfficientNetV2B0	0.9815	0.9815	0.9817	0.9815	11.624
EfficientNetV2S	0.9827	0.9827	0.9830	0.9827	25.682
InceptionResNetV2	0.9225	0.9224	0.9226	0.9225	19.072

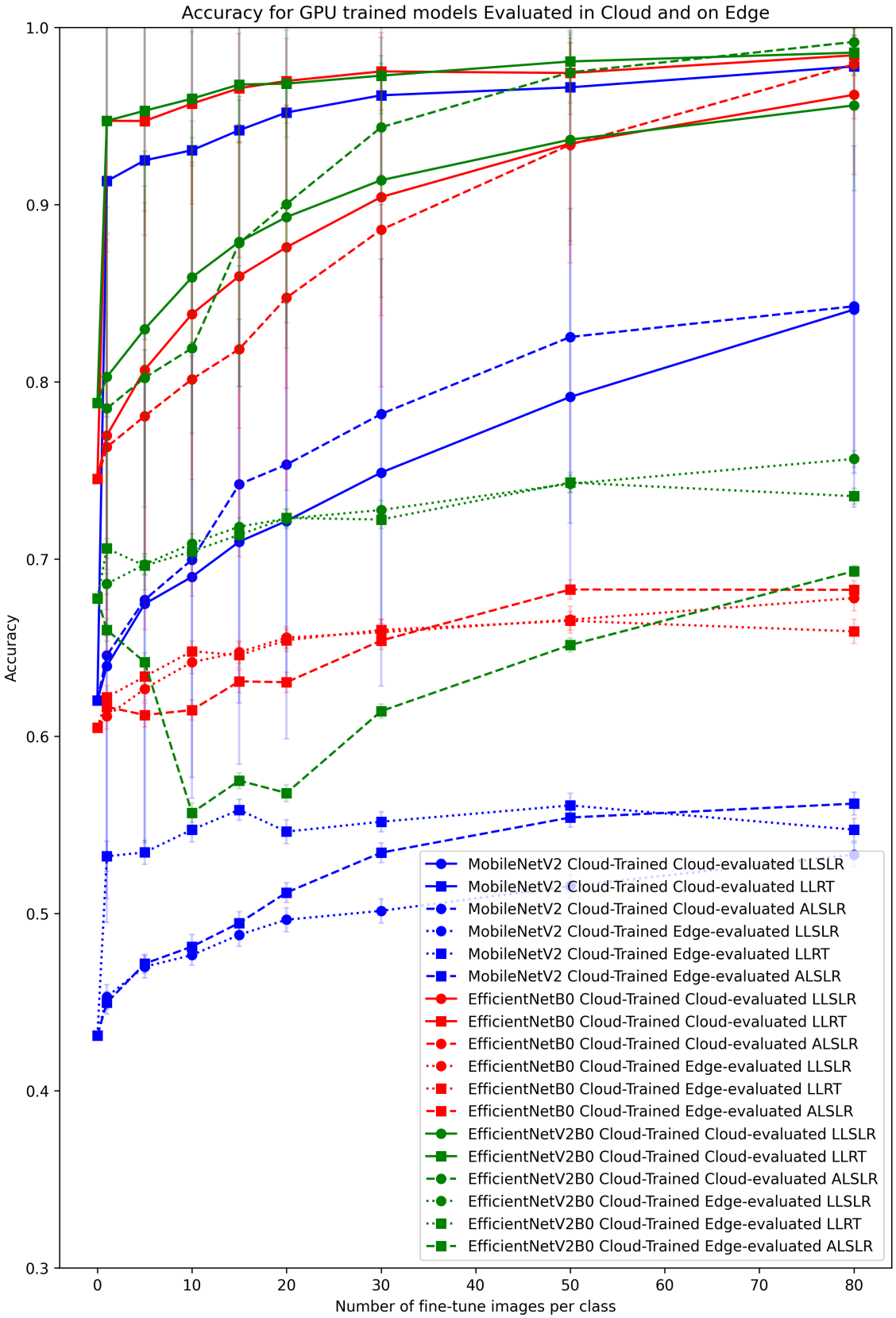


Figure 52: The accuracy for the varying amounts of OOD data for fine-tuned models that are trained on the Cloud and tested on both Cloud and Edge to investigate the impact of testing on different platforms. The accuracy of 0 images per class is given by the non-fine-tuned models as given in experiment 3.

D RESULTS EXPERIMENT 5

D.1 *Classification Performance*

Table 25: The accuracy and F1 score of the fine-tuned models in the cloud and on the edge.

Algorithm	1	5	10	15	20	30	50	80
MobileNetV2								
GPU LLSLR on Edge	0.4530	0.4700	0.4765	0.4880	0.4965	0.5015	0.5155	0.5332
GPU ALSLR on Edge	0.4497	0.4717	0.4813	0.4947	0.5118	0.5343	0.5542	0.5620
GPU LLRT on Edge	0.5323	0.5345	0.5473	0.5585	0.5462	0.5518	0.5610	0.5473
EdgeTPU LLRT	**0.5853**	**0.5922**	**0.6115**	**0.6153**	**0.6155**	**0.6273**	**0.6565**	**0.6653**
EfficientNetB0								
GPU LLSLR on Edge	0.6113	0.6268	0.6418	0.6475	0.6557	0.6587	0.6658	0.6780
GPU ALSLR on Edge	0.6167	0.6120	0.6148	0.6310	0.6305	0.6540	0.6828	0.6827
GPU LLRT on Edge	0.6220	0.6337	0.6480	0.6458	0.6542	0.6600	0.6652	0.6592
EdgeTPU LLRT	**0.7878**	**0.7992**	**0.8008**	**0.8050**	**0.8158**	**0.8288**	**0.8393**	**0.8498**
EfficientNetV2B0								
GPU LLSLR on Edge	0.6860	0.6970	0.7088	0.7183	0.7228	0.7277	0.7425	0.7565
GPU ALSLR on Edge	0.6600	0.6418	0.5568	0.5750	0.5680	0.6142	0.6515	0.6932
GPU LLRT on Edge	0.7060	0.6963	0.7043	0.7137	0.7232	0.7223	0.7432	0.7355
EdgeTPU LLRT	**0.8058**	**0.8113**	**0.8158**	**0.8192**	**0.8205**	**0.8295**	**0.8400**	**0.8603**

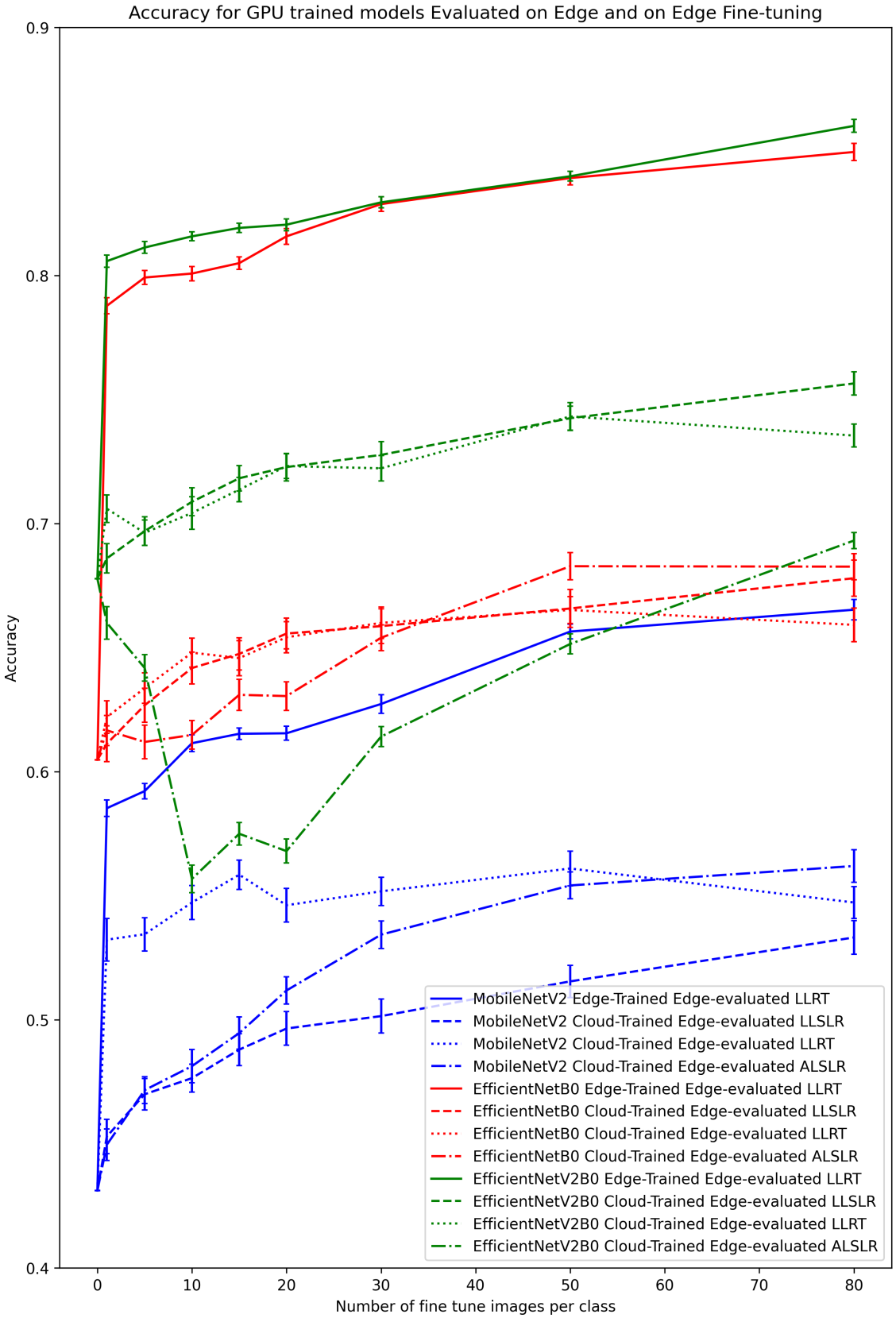


Figure 53: The accuracy for the varying amounts of OOD data for fine-tuned models that are trained on the Cloud and tested on Edge, as well as the models fine-tuned on the Edge. The accuracy of 0 images per class is given by the non-fine-tuned models as given in experiment 3.

D.2 *Power Consumption*

Table 26: The average time in seconds required per fine-tuning for each edge TPU model.

Algorithm	Power (W)	Time (s/model) for number of fine-tuned images per class							
		1	5	10	15	20	30	50	80
MobileNetV2	3.439	101.45	101.274	101.904	102.514	103.129	104.437	106.97	110.758
EfficientNetB0	3.595	173.427	173.067	174.495	175.757	177.104	179.713	184.996	193.038
EfficientNetV2B0	3.706	194.135	194.635	196.08	197.642	199.199	202.159	208.214	217.435

Table 27: The average power in Joule required per fine-tuning for each edge TPU model.

Algorithm	Power (W)	Power (J/model) for number of fine-tuned images per class							
		1	5	10	15	20	30	50	80
MobileNetV2	3.439	348.887	348.281	350.448	352.546	354.661	359.159	367.87	380.897
EfficientNetB0	3.595	623.47	622.176	627.31	631.846	636.689	646.068	665.061	693.972
EfficientNetV2B0	3.706	719.464	721.317	726.672	732.461	738.231	749.201	771.641	805.814

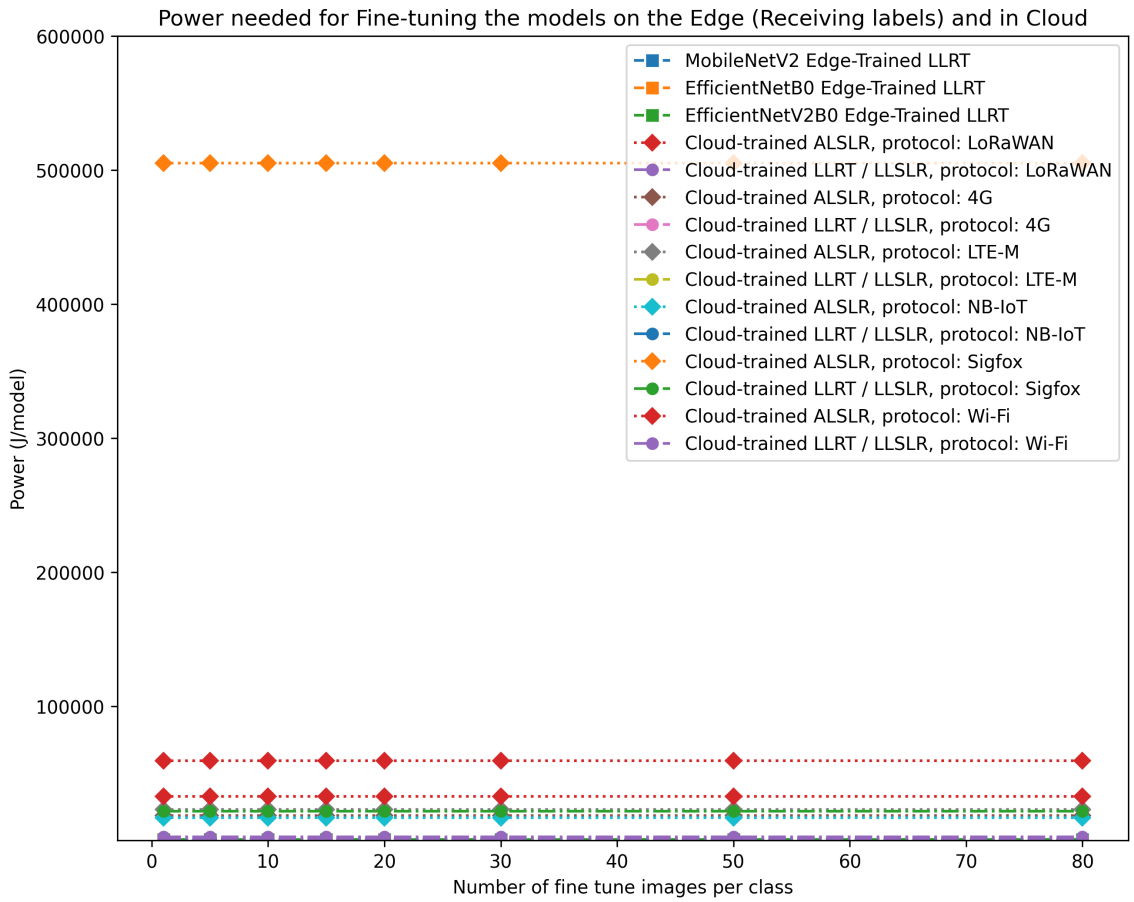


Figure 54: The power consumption in Joule per cloud and edge model, which is consumed for fine-tuning. The protocols are shown for images of 36.0kB.

Table 28: Here the cloud fine-tuned models are shown. The average Power in Joule needed per fine-tuning a model is shown. This is needed by the edge device for receiving the updated weights for the fine-tuning techniques LLRT, LLSLR and ALSLR.

Algorithm	Power (J/model) for sending the weights to the edge
LoRaWAN	
Cloud trained ALSLR	59693.10
Cloud trained LLRT / LLSLR	2568.35
4G	
Cloud trained ALSLR	18565.14
Cloud trained LLRT / LLSLR	798.78
LTE-M	
Cloud trained ALSLR	23093.10
Cloud trained LLRT / LLSLR	993.60
NB-IoT	
Cloud trained ALSLR	16934.94
Cloud trained LLRT / LLSLR	16934.94
Sigfox	
Cloud trained ALSLR	505161.56
Cloud trained LLRT / LLSLR	1735.00
Wi-Fi	
Cloud trained ALSLR	32929.05
Cloud trained LLRT / LLSLR	1416.80

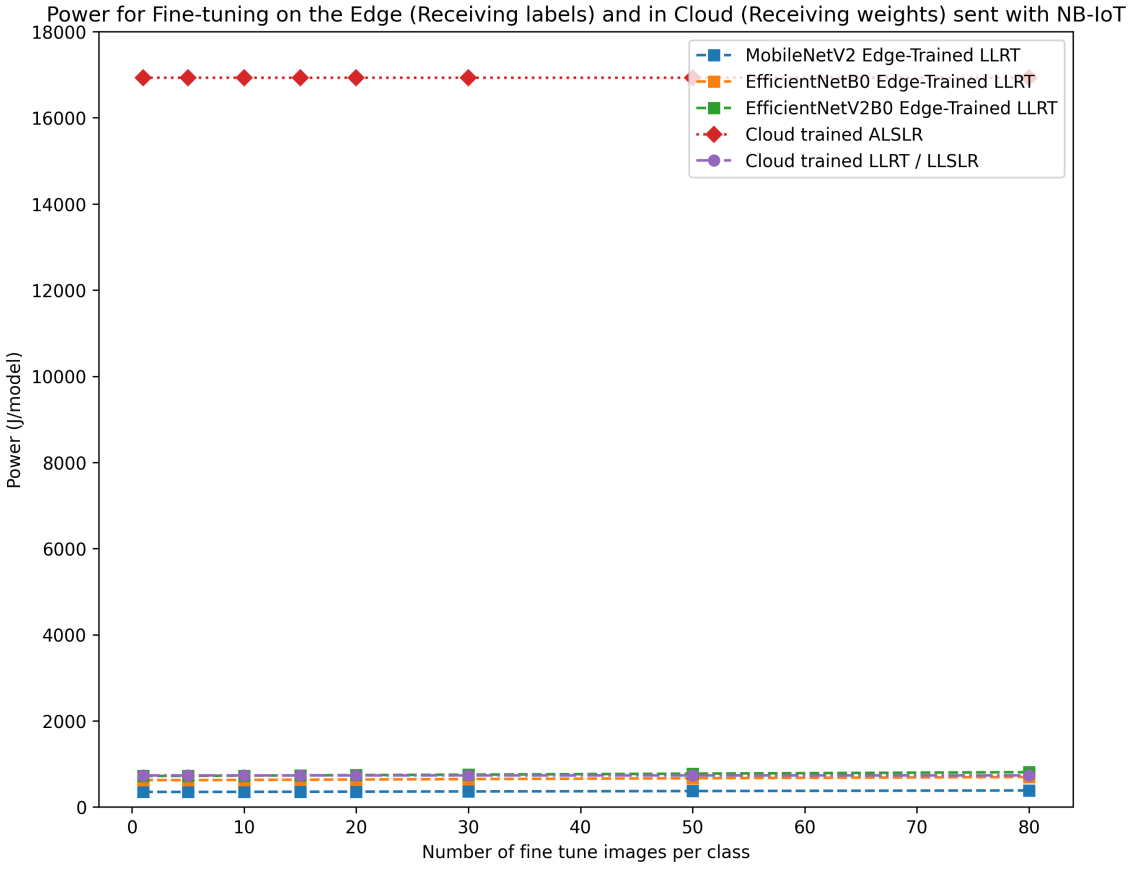


Figure 55: The power consumption in Joule per cloud and edge model, which is consumed for fine-tuning. The protocol which is most power efficient is shown.