TUM Data Innovation Lab

Munich Data Science Institute

Technical University of Munich

&

SigOpt (intel)

&

Leibniz-Rechenzentrum (LRZ)

Final report of project:

How To Train the Best Deep Learning Models for The Edge?

Authors	Rayen Dhahri, Nicolas Gehring, Grigor Nalbandyan Thejeshwar Sai, Juergen Wallner
Mentor(s)	Maja Piskac (M.Sc) LRZ, Tobias S Andreasen (M.Sc) SigOpt
Project Lead	Dr. Ricardo Acevedo Cabra
Supervisor	Prof. Dr. Massimo Fornasier

Aug 2022

Abstract

Artificial Intelligence (AI) applications and computations on limited endpoint resources, referred to as edge devices, play a crucial role in the massively growing domain of edge computing. Due to the limited hardware environment of edge devices, it requires special techniques and methods to efficiently run or even fit models on such devices. Therefore, a thorough literature research on specific techniques like sparsification and quantization as well as a detailed explanation about hardware limitations is presented. This knowledge is further used to design and implement an experimentation pipeline to study the impact of the sparsification and quantization methods on the model's performance, as well as hardware extension effect on inference speed. We show the effect of hyperparameters on multiple training metrics as well as showcase how different input sizes, batch sizes, and CPU cores influence the inference speed. The pipeline was developed on the AI infrastructure hosted and operated by the "Leibniz-Rechenzentrum (LRZ) der Bayerischen Akademie der Wissenschaften (BAdW)" using the Neural Magic Ecosystem, a widely used open-source sparsification framework. To make our results comparable to current publications, in all experiments, a ResNet-50 architecture on the image classification dataset CIFAR-100 was used. Overall, we were able to obtain a four times more compact and five times quicker model with above baseline accuracy, thanks to fine-tuned sparsification and quantization approaches. We deploy the top performing models on several Edge devices.

Keywords: Edge devices, sparsification, quantization, hyperparameter tuning, deep learning models

Contents

Abstract **1**

1 Introduction **4**
1.1 Motivation . . . 5
1.2 Challenges . . . 6
1.3 Approach and Contribution of this Work . . . 6

2 Theoretical Background **8**
2.1 Sparsification . . . 8
2.1.1 What to Sparsify? . . . 9
2.1.2 When to Sparsify? . . . 9
2.1.3 How to Sparsify? . . . 10
2.2 Quantization . . . 11
2.3 Reduced Peak Memory Consumption . . . 11
2.4 Hardware Limitations . . . 13

3 Experiment Architecture and Implementation **14**
3.1 Conceptualization Phase . . . 14
3.2 Planning and Implementation Phase . . . 16
3.2.1 Model Modification . . . 17
3.2.2 Benchmarking . . . 18
3.2.3 Metric Collection . . . 18
3.2.4 Optimization . . . 18

4 Experiments **20**

5 Results **21**
5.1 Effect of Sparsity Level on Accuracy and Inference Speed . . . 21
5.2 Effect of Batch Size and Number of Cores on Inference Speed . . . 21
5.3 Effect of the Input Size on Inference Speed . . . 22
5.4 Effect of Quantization on Inference Speed . . . 23
5.5 Effect of Recipe Hyperparameters on Train Metrics . . . 24
5.6 Model Deployment on different Edge Devices . . . 26
5.7 System Resources Usage . . . 27

6 Conclusion and Further Research **29**

References **30**

List of Figures

1 Edge Devices and Nodes in Relation to the Cloud 4
2 Edge Computing Interest (Google Trends) 5
3 Edge Computing Interest per Region and Ranking (Google Trends) 5
4 Typical Test Error vs. Sparsity showing Occam's Hill (network: ResNet-50 on Top-1 ImageNet) 8
5 Peak Memory Usage before Operators Reordering in a simple CNN Model [15] 12
6 Peak Memory Usage after Operators Reordering in a simple CNN Model [15] 12
7 Overview of our System Architecture 17
8 Effect of Sparsity on Inference Speed 21
9 Effect of Batch Size and Number of Cores on Inference Speed 22
10 Input Size and Latency (Batch Size 1) 22
11 Input Size and Latency (Batch Size 16) 23
12 Quantization Effect (Batch Size 1) 23
13 Quantization Effect (Batch Size 16) 24
14 Pruning Fraction and Pruning Frequency Effect on the Validation Accuracy 25
15 SigOpt Parallel Coordinates of Recipe Parameters on Validation Accuracy 25
16 Comparison of Models Deployed on Edge Device 26
17 Comparison of Models' Memory Footprint on Edge Devices 27
18 GPU Power Usage (left)/ GPU Memory Usage (right) for only Sparsification and Sparsification with Subsequent Quantization 28

List of Tables

1 Hardware Limitations. 13
2 DeepSparse CPU Hardware Support 16
3 Used Edge Devices Specifications 26

1 Introduction

With the breakthrough of Artificial Intelligence (AI), there is an ever-increasing boom of applications and services. Benefiting from this technology, our lifestyles have been dramatically changed accordingly. [26] Logically, it is estimated that by 2025 more than 75 billion devices [12], such as smartphones, tablets, wearables, and gadgets, we refer to as edge devices (classification of terms in Figure 1), will interact with their surrounding environment and user. The world is entering the age of hyperconnectivity, where data and information systems share data between and among them constantly. [23]

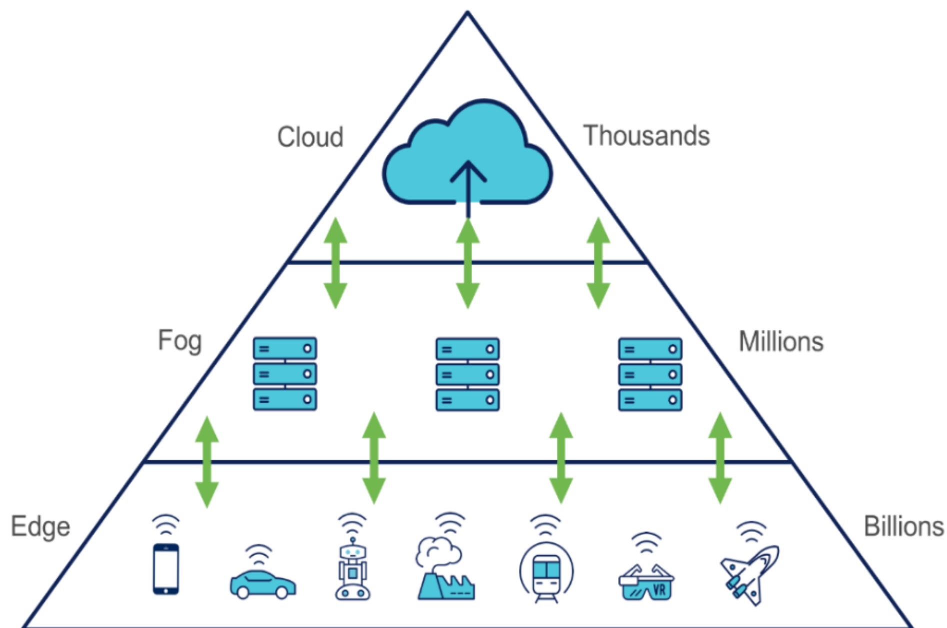


Figure 1: Edge Devices and Nodes in Relation to the Cloud
Taken from OCTO Technology

This high level of connected things will lead to network congestion eventually. According to Cisco's forecast, there were 850 ZB of data generated by mobile users and Internet of Things (IoT) devices in 2021 [26], [3], making cloud computing insufficient, since it suffers from delayed data transfer, coherent communication costs and exposed private information.

As preventive measurement, understanding the complex analysis of sensors data for decision-making as close to the source as possible is inevitable. Therefore, edge computing is a rapidly growing and developing field as reflected in Figure 2 and 3.

Figure 2: Edge Computing Interest (Google Trends)

Figure 3: Edge Computing Interest per Region and Ranking (Google Trends)

Emphasizing figures from Gartner, a tech research and advisory firm, show that approximately 10% of enterprise-generated data was created and processed outside traditional centralized data centers and the cloud in 2018. But it predicted that 75% of data will be processed at the edge by 2025. [25] The sole usage of IoT devices, however, face also several challenges in order of this accomplishment. Necessarily, the subject and comprehensive approaches must be motivated particularly.

1.1 Motivation

Varghese et. al. [24] identified a list of needs that motivate computing on edge devices. Edge computing takes the computation out of a data center and places it closer to endpoint devices where data is being generated, which brings several benefits to approach such needs [23]:

1. Improved speed / reduced latency: Computation closer to the source improves the services by solving the latency challenge of current cloud infrastructures connected to edge devices, because it eliminates the need to move data from endpoints to the cloud and back again. This is especially helpful in autonomous vehicles and medical use cases, which all require analysis and instructions nearly instantaneously in order to function safely.

2. Improved security and privacy protections: Edge devices capture confidential sensory input like text, audio, video, touch, or motion. Without the need of data transfer also comes the benefit that data is kept close to the edge, which provides enhanced security and more privacy protection to the end users.

3. Savings / reduced costs: In the next decade, data centers will most likely consume more than three times as much energy as today due to the ever-rising amount of data. Hence, there is an unavoidable need for adopting energy efficient strategies in the form of performing analytical approaches on edge devices that can minimize this energy usage. Moreover, users need more bandwidth to cope with this amount, further driving up the prices. Without or limited data movement, edge computing can help to keep the costs moderately.

4. Reliability and resiliency: One of the biggest advantages of operating edge devices is, that they continue to function when communication channels are slow or temporarily down. Additionally, it helps to reduce central point of failures, since a failure at one edge device will not affect the performance of other ones in the ecosystem.

5. Scalability: Business-wise, companies can benefit from the deployment of edge devices in different fields of application.

1.2 Challenges

However, existing intelligent applications are computation intensive, which present strict requirements on resources, e.g., CPU, GPU, memory, and network [26]. This makes it especially hard to deploy deep learning models on edge devices. Additionally, tasks rely heavily on their domain. Different metrics are required for particular cases, which makes the optimization even more complex [20]. Below are some essential metrics to consider when planning to run software on edge devices.

1. Latency: The goal is to finish a given workload in the nearest layer which has enough computation capability. Since edge devices are very limited in their resources (see also 2.4) it is naturally challenging.

2. Energy: User services rely on functional devices, which means that any unnecessary battery drainage is problematic. Hence, it can be considered as the most precious resource for edge devices.

3. Cost: Respective analysis needs to be done in runtime, but should also consider inference and resource usage of concurrent workloads.

1.3 Approach and Contribution of this Work

With all these considerations in mind, our work on this project was guided by the following three research questions:

RQ1: How to deploy Deep Learning models efficiently on edge devices from the literature standpoint?

This research question is intended to build the theoretical foundation about hardware limitations of edge devices and methods to overcome them. By structurally organizing and evaluating related literature, an overview of relevant approaches is created.

RQ2: How can the impact of those approaches be tested, optimized and visualized?

This research question derives requirements for an experimentation pipeline, which can be used to test the methods from RQ1. Further, the pipeline must allow the optimization and the visualization of the impact of different methods. The creation of such a pipeline is delivered as an artifact in the context of this question.

RQ3: How do different methods and their hyperparameter settings perform with respect to different metrics in the training and inference stages of a computer vision model?

This research question focuses on performing actual experiments using the pipeline from RQ2. The aim is to understand the influence of different factors, such as hyperparameters, chosen method or hardware infrastructure on various train or inference metrics.

This report follows the three above-mentioned research questions while presenting and explaining all steps taken by the authors to answer them.

Section 2 gives detailed insight into the approaches to transform or train deep learning models for edge devices, as well as an overview of current edge device hardware products and their limitations.

Section 3 describes the development of the experiment pipeline. The conceptual work, and all used frameworks, are explained. Further, a technical description of the software architecture is provided and the implemented features and design choices are explained in detail.

Section 4 explains the actual conducted experiments. It motivates the individual runs by explaining the underlying questions we wanted to answer and shows how they were mapped to specific configurations for particular experiments.

Section 5 presents the results of the experiments. Qualitative and quantitative results are shown and discussed.

Section 6 closes with a conclusion and discusses limitations of the work as well as providing possible further research directions.

2 Theoretical Background

In order to utilize the previously mentioned advantages of edge computing and in particular AI models on edge devices, new and innovative approaches are essential. These methods aim to modify deep learning models in such a way that they can be used efficiently despite the restrictions of the limited hardware environment. For example, existing models can be reduced in size or small models can be trained from the beginning to fit on a particular device. As size is a big issue, most methods focus on the reduction of model size and latency. Therefore, in the next two subchapters, promising approaches to reduce model size called sparsification and quantization will be presented. Because not all methods deal with the reduction of model size, we also present an approach that reduces peak memory consumption by intelligently reordering the individual instructions. In order to clarify the more challenging hardware environment of edge devices, an additional chapter follows, which explains more precisely why it is difficult to deploy models on edge devices. Different hardware types are presented and useful hardware extensions for the deep learning context are discussed.

2.1 Sparsification

Sparsification describes the selective pruning of neural network components with the goal of improving generalization and robustness, as well as improving performance for inference and/or training while reducing model size. It builds on the assumption that not all features/components of a network are equally relevant and aims to represent dense vectors/networks in sparse subspaces. Today's sparsification methods can lead to a 10-100x reduction in model size, and to corresponding theoretical gains in computational, storage, and energy efficiency, all without significant loss of accuracy [11]. In most cases, the accuracy follows Occam's hill [18] , which can be seen as the green line in Figure 4.

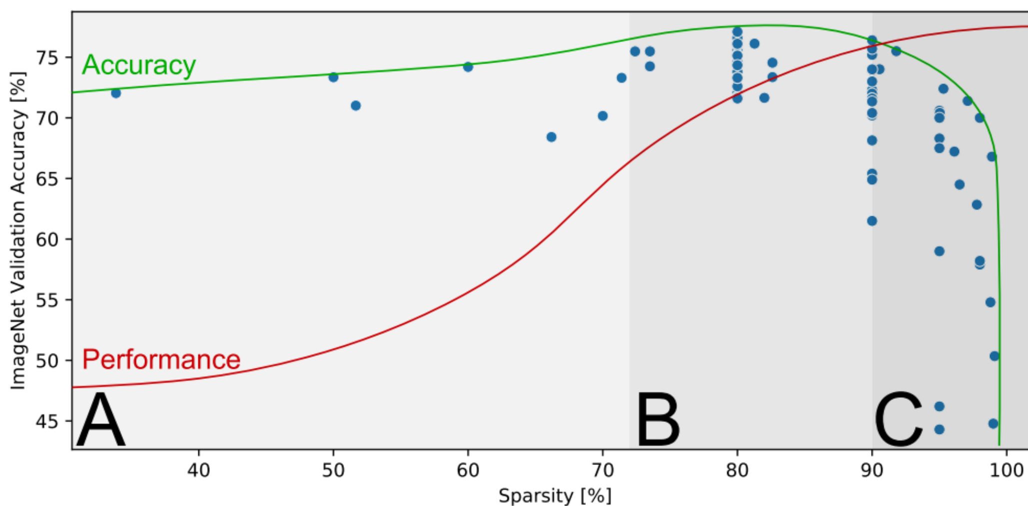


Figure 4: Typical Test Error vs. Sparsity showing Occam's Hill (network: ResNet-50 on Top-1 ImageNet)

Taken from [11]

The picture shows three main stages of the behavior of a sparsified model in terms of accuracy and performance. In low sparsification regimes (A) the accuracy initially increases because smaller models form a stronger regularizer, which enforces the model to focus on more important and general aspects and therefore decreases learned noise. The performance on the other hand grows slowly due to overhead introduced by additional data structures to store the sparsified model and control sparse computations. In the second regime, with higher sparsity levels (B) the model performance remains stable and perhaps slightly decreases, while performance increases faster. Finally, at high levels of sparsity (C), quality drops rapidly while performance levels off due to storage and control overhead dominating performance.

The concept of sparsification can be applied to different elements of a network, at different points in time, by using a huge variety of criteria to select the elements for removal. The three aforementioned aspects (what to sparsify, when to sparsify and how to sparsify) are important considerations which will be further described in the next paragraphs.

2.1.1 What to Sparsify?

When considering which elements to sparsify one distinguishes between model and ephemeral sparsity. The first one introduces sparsity for a whole model targeting weights, neurons or neuron-like elements like filter, channels or heads in other architecture types. The latter is a second class of sparsification approaches, which are only applied to and relevant for single examples. These techniques include structural sparsity introduced by well-known activation functions like the ReLu or Softmax which naturally push values to zero and thereby introduce sparsity. One can also consider random activation dropouts [21] as a way of introducing ephemeral sparsity. Sparsification applied to gradients and/or errors, as well as routing each example through a different sparse subnetwork, are other examples.

A way dropout is used to achieve a high compression ratio is described in the DeepIoT [27] method. The main approach of DeepIoT can be summarized as a set of dropout operations between the layers with an optimal dropout probability for each hidden element in the network. The dropout probabilities are determined through a compression neural network that is jointly optimized with the original network through the compressor-critic framework, where we minimize the same loss function based on the original network parameter and the weights that are given to the compression network as an input. This joint simulation allows the original neural network to be optimized to achieve a better performance and allows the compression network to produce better dropout probability.

2.1.2 When to Sparsify?

After deciding what to sparsify it is also important to consider when to sparsify the respective components. One can categorize the different methods into three main schedules. The most common choice is to first train a model and then sparsify it. This means to train a dense model until convergence and sparsify it afterwards. One can distinguish between methods which try a one-shot approach and others which include retraining and iterative sparsification processes. Those methods usually consist of several stages (e.g. warm-up, iterative sparsification, retraining or fine-tuning) as explained in [22, 11].

Another option is to sparsify a model during the training process. This option is usually cheaper as the first schedule because a sparse model is less computationally expensive. Furthermore, it could prevent dense models which have been trained to convergence from overfitting, which would be hard to recover from with pruning alone. Yi [22] showed that with their dense-sparse-dense training scheme they reach significantly higher generalization performance for their computer vision task. The schedules usually also include multiple iterations consisting of different training phases. The main challenge with those approaches is to decide how fast to prune how many elements. These problems and the fact that they are more brittle to hyperparameter configurations and could potentially lead to less efficient convergence [9] makes this approach harder in practice.

The last schedule of fully sparse training starts with a sparse model and trains it in the sparse domain. One can differentiate between static and dynamic sparsity. Dynamic sparsity combines pruning and regrowth of elements during the training process, while static sparsity removes elements once before the training starts and does not update them again [11]. Although these methods require complex schedulers and hyperparameter settings, they enable the training of models which would normally just not fit into the machine. Further, they require less resources and therefore save costs for training. This raises the question on how to pick the initial sparse model, as show in [13, 5, 6], where randomly initialized sparse networks perform much worse than their dense counterparts. There exist multiple actively researched approaches for finding those initial sparse subnetworks which can reach baseline accuracy. An interesting example is the research around the Lottery Ticket Hypotheses [6] which claims that "A randomly-initialized, dense neural network contains a subnetwork that is initialized such that - when trained in isolation - it can match the test accuracy of the original network after training for at most the same number of iterations." These hypotheses have led to many further developments and proves that those subnetworks exist in transformer models [1], large computer vision models [2] and that they can be used for reinforcement learning [28] too.

2.1.3 How to Sparsify?

After it is clear what and when to prune elements of the networks, one needs to consider how the elements for removal are actually determined. For this purpose there exists numerous methods as shown in the overview of [11]. The most important takeaway is that comparative studies failed to identify a clearly winning method as the efficiency of a method depends on the exact setting of network architecture, hyperparameters, learning rate schedule, learning task etc. [7]. However, all methods aim to derive an importance measurement for the objects which are considered for removal. The first class of methods are data-free approaches. Those sparsify without considering training examples or evaluating the network. An example would be to prune weights by magnitude [4]. The second class of data-driven approaches contains methods which use a set of examples (potentially all the training data) to determine directly which elements should be removed to maintain or improve prediction accuracy while sparsifying the network. The last class of training-aware approaches consider the training loss function itself in the pruning process and use it to improve the model accuracy of the pruned network as much as possible. Examples would be methods which are using the first or second order Taylor expansion of the Loss

function to derive weight importance, or include sparsification goals in the Loss function as a regularization term.

2.2 Quantization

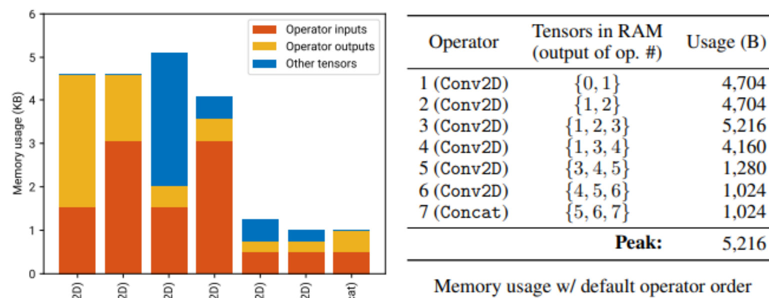
Quantization is another method for addressing the issues of latency, memory footprint and energy consumption in neural networks [8]. With its roots in Shannon's mathematical theory of communication [19], currently quantization is a widespread and efficient approach used both in theoretical works, and also in practical applications. Quantization reduces the precision of neural network weights, input, and output activations from 32-bit floating-point representation to lower precision, usually 8 bits. This results in faster inference, memory footprint reduction of four times, smaller parameter updates, higher cache utilization, etc. Recent advances of modern hardware architectures enable us to experience the theoretical gains in practice. The main downside of quantization is the possible degradation of accuracy metrics, which can be fixed by Quantization-Aware Training (QAT) or Post-Training Quantization (PTQ). In QAT, a model is quantized and fine-tuned on training data, which allows the model to restore its accuracy by adjusting the parameters. PTQ uses a calibration data to gather statistics about model's layers' input and output distributions and perform Int8 conversion accordingly. Although PTQ does not require additional retraining, it can produce lower accuracy compared to QAT [8].

2.3 Reduced Peak Memory Consumption

We have discussed in detail the techniques like sparsification and quantization that are used for deploying a neural network model on edge devices. Though they focus on reducing the complexity of the network, we should also look out at the memory limitations of an edge device and how it affects neural network deployment. This section discusses on one such technique to minimize the peak memory usage of a neural network by making inference to follow a particular execution order of its operations reducing the memory footprint enough to make it fit within the on-chip memory of our edge platform, which would otherwise not have been possible using the default provided operator execution order.

Modern deep learning frameworks optimize the network's computation graph for inference in advance by fusing adjacent operators and folding batch normalization layers into preceding linear operations. An operator requires buffers for its inputs and output to be present in memory before its execution can commence. This comprises input and output tensors of a pending operator and other tensors that were already produced and need to be held back in memory for subsequent operators. However, more recent architectures, such as ResNet, Inception, NasNet, introduce divergent processing paths where the same tensor can be processed by several layers, i.e. their computation graph is no longer linear and has branches. Neural networks whose computation graphs comprise branches permit little freedom over the order of assessment in their operators. When execution reaches a branching point, the inference software program has to pick which department to begin comparing next. This desire can have an effect on which tensors want to be stored in rem-

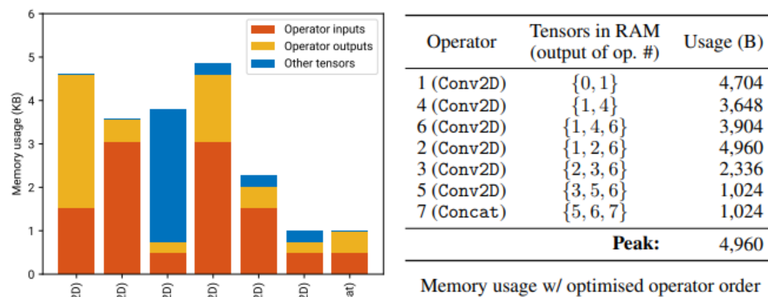
iniscence (running set), so we are able to assemble an execution timetable that minimizes the overall length of the running set at its peak (reminiscence bottleneck) [15]. In a nutshell, changing the evaluation order of a model's operators minimizes peak memory usage of a neural network during inference. Figure 5 and Figure 6 showcases the differences in the memory footprint of a simple CNN model before and after reordering.The model's peak memory consumption is reduced by 256 bytes [15]. Though this doesn't change the inference speed, but the model is optimised in terms of memory. Results based on this approach is discussed later part in this report. Employing a different operator execution order for neural network inference can make previously non-deployable models fit within the memory constraints of an edge hardware. Unlike mobile and server platforms, edge hardware often does not have enough memory to statically pre-allocate all tensor buffers of the network, which requires the inference to support dynamic memory allocation. Edge devices like microprocessors and microcontrollers are a viable platform for running deep learning applications if the model designer can overcome constraints imposed by limited memory and storage.



Operator	Tensors in RAM (output of op. #)	Usage (B)
1 (Conv2D)	{0, 1}	4,704
2 (Conv2D)	{1, 2}	4,704
3 (Conv2D)	{1, 2, 3}	5,216
4 (Conv2D)	{1, 3, 4}	4,160
5 (Conv2D)	{3, 4, 5}	1,280
6 (Conv2D)	{4, 5, 6}	1,024
7 (Concat)	{5, 6, 7}	1,024
	Peak:	5,216

Memory usage w/ default operator order

Figure 5: Peak Memory Usage before Operators Reordering in a simple CNN Model [15]



Operator	Tensors in RAM (output of op. #)	Usage (B)
1 (Conv2D)	{0, 1}	4,704
4 (Conv2D)	{1, 4}	3,648
6 (Conv2D)	{1, 4, 6}	3,904
2 (Conv2D)	{1, 2, 6}	4,960
3 (Conv2D)	{2, 3, 6}	2,336
5 (Conv2D)	{3, 5, 6}	1,024
7 (Concat)	{5, 6, 7}	1,024
	Peak:	4,960

Memory usage w/ optimised operator order

Figure 6: Peak Memory Usage after Operators Reordering in a simple CNN Model [15]

2.4 Hardware Limitations

Using deep learning models directly on edge devices allows for greater cost-efficiency, scalability and privacy for end users, compared to relying on a remote server to carry out the processing. However, an edge unit typically consists of a low-frequency processor and only several hundred kilobytes of on-chip memory, and thus are severely under powered compared to mobile devices. Specially designed network architectures and inference software are required to cope with hardware constraints of edge devices. The below table 1 showcases the different edge devices and their respective support to Tensorflow lite (Tflite). We could see that most of the devices don't support the Tflite software.

End Device	DNN Model	Application	TFlite Support	Key Metric
STM32F401RE	CNN	Image Recognition	no	fast inference
Raspberry Pi model 4	SVM	Image Recognition	no	fast inference
Raspberry Pi model 4	DNN	Distributed Computing	no	hierarchical
Raspberry Pi model 4	SVM, CNN	Video Analysis	no	fast inference
Raspberry Pi model 4	SVM	Video Analysis	no	fast inference
SPHERE	SVM	Battery Lifetime Estimation	no	energy
Motorola 68HC11	CNN	Image Recognition, Sensor Fusion	no	fast inference
ARMÂ® v7	SVM	Code execution	no	accuracy
Sparkfun Edge	LR	Speech Recognition	yes	accuracy
Arduino Nano 33 BLE Sense	-	-	yes	-
Adafruit EdgeBadge	-	-	yes	-
ESP32	CNN	Human Activity Recognition	yes	accuracy
Espressif ESP32-DevKitC	-	-	yes	-
Espressif ESP-EYE	-	-	yes	-
Wio Terminal: ATSAMD51	-	-	yes	-
Sony Spresense	-	-	yes	-
Google Edge TPU	CNN	-	yes, with limits	fast inference

Table 1: Hardware Limitations.
Taken and extended from [16].

Edge computers are great for collecting, storing, processing, and analyzing data at the edge; however, for some complex workloads, edge computers should be equipped with performance accelerators for real-time processing decision-making. New computing and storage designs maximize performance as close to the data as possible. These add-in hardware solutions through PCIe architecture provide added value for specific edge computing workloads that require performance for real-time processing. Nevertheless, power source and utilization must be managed in such a way that the edge device is up all the time.

3 Experiment Architecture and Implementation

After establishing a solid theoretical foundation around the challenges and approaches to bring deep learning models onto edge devices, our goal was to practically explore some of them. We decided to try some of the most promising sparsification and quantization approaches and do experiments around their performance and optimization before deploying them on real edge devices and benchmarking their performance.

Our aim was to build a pipeline which allows a user to test different sparsification and quantization approaches on a specific model and dataset. Further, the pipeline should collect metrics of the whole training/sparsification process, as well as the inference of the altered models for later analysis. In order to compare and optimize hyperparameter settings, the option to apply multiple configurations automatically as well as use multiple approaches for their optimization must be included. The development of the experimentation architecture proceeded in four phases. First, there was a conceptual phase. Based on the above-mentioned requirements, we examined which open source software as well as hardware provider could we use for the creation of our experimentation architecture. This was followed by the planning phase, in which the architecture was outlined using the before determined software and resources.

Subsequently, the actual implementation and setup followed and by the end we benchmarked and deployed these models on real edge devices. In the next section, the conceptual phase is described first. Afterwards, planning and implementation are described together. The conduction of the actual experiments using this system is explained in 4.

3.1 Conceptualization Phase

In the conceptual phase, we derived the most important requirements for our system and searched for the most promising open source software and tools to fulfill them. In the following, our four main requirements are shown and the tools chosen to address them are presented.

Requirement 1: The pipeline can apply different methods for sparsification and quantization to a given deep learning model and dataset.

After researching and evaluating several open source sparsification frameworks, we decided to use the Neural Magic Ecosystem [1]. Founded by a team of award-winning MIT computer scientists, Neural Magic is the creator and maintainer of the DeepSparse Platform. It has several components, including the DeepSparse Engine, a CPU runtime that runs sparse models at GPU speeds, as well as three other products called Sparsify, SparseML, and the SparseZoo. We used all of their offered software solutions except Sparsify. SparseML is a toolkit that includes APIs, CLIs, scripts and libraries to apply state-of-the-art sparsification algorithms to any neural network.

[1] https://neuralmagic.com/

The configuration of SparseML is done in the form of a recipe. A recipe is a YAML file which describes the so called modifiers which should be applied to the model. There exist multiple modifiers for tasks such as pruning, learning rate adaption or quantization. Those modifiers can be configured as well to further optimize the performance. An example of such a recipe depicting some hyperparameters and modifiers can be seen in Listing 1.

To create custom recipes, we made wide use of SparseZoo, which is Neural Magics constantly-growing repository of sparsified models with matching sparsification recipes. In order to run and benchmark our sparsified models, we decided to use the DeepSparse Engine.

Listing 1: Example Recipe

```
1  version: 0.1.0
2  final_sparsity = 0.95
3  modifiers:
4      - !EpochRangeModifier
5          start_epoch: 0.0
6          end_epoch: 70.0
7
8      - !LearningRateModifier
9          start_epoch: 0
10         end_epoch: -1.0
11         update_frequency: -1.0
12         init_lr: 0.005
13         lr_class: MultiStepLR
14         lr_kwargs: { milestones : [43, 60],  gamma : 0.1}
15
16     - !GMPruningModifier
17         start_epoch: 0
18         end_epoch: 40
19         update_frequency: 1.0
20         init_sparsity: 0.05
21         final_sparsity: eval(final_sparsity)
22         mask_type: unstructured
23         params: [ sections.0.0.conv1.weight ,
24          sections.0.0.conv2.weight ,
25          sections.0.0.conv3.weight ]
```

Requirement 2: The pipeline can create and report meaningful metrics for the training/sparsification process as well as the inference of the optimized models.

As our main source of metrics collection for both training and inference, we decided to use Weights & Biases (wandb)[2]. Wandb is an experiment tracking tool for machine learning with a quite flexible and customizable API. We used it to track all our experiments due to its functionality to report custom metrics and log parameters as well as their great visualization capabilities.

[2] https://wandb.ai/

Requirement 3: The model modification process can be tuned via hyperparameters and includes optimization software to derive an optimal configuration.

To meet this requirement, we have decided that our software must be able to accept multiple different recipe configurations for a single run and automatically apply them one after another without manual intervention. Further, for each of these different configurations, the metrics must be automatically reported in a way that we can later analyze the effect of the individual changes. Only in this way, we can test hundreds of different configurations. In addition, we tried SigOpt [3], a hyperparameter optimization tool, as an alternative approach to derive an optimal configuration besides manual grid search.

Requirement 4: A powerful and versatile hardware infrastructure supports training and inference.

As most of the sparsification and quantization approaches provided by SparseML require retraining, fine-tuning and further computations, we need a hardware infrastructure to train and modify our models. For this purpose we got granted access to the Leibniz-Rechenzentrum (LRZ) AI Systems[4] which is one of the LRZs[5] services primarily oriented towards big data and AI communities with a focus on GPU resources. The AI systems consist of a total of 6 different partitions with different hardware resources. For our experiments we choose the V100 GPU Nodes which are equipped with NVIDIA Tesla V100 16 GB GPUs [6], 20 CPUs and 368 GB of RAM. For the inference benchmarking of our modified models we decided to use Google Cloud as their Compute Engine[7] service and especially their N2 machines give us access to CPUs with Vector Neural Network Instructions (VNNI) which are optimized for deep learning inference as show in table 2. Further, SparseML also recommends in their documentation to use such resources to achieve optimal results. In addition, we deployed the models on several private smaller edge devices like a Raspberry Pi 3, Raspberry Pi 3 B+, Raspberry Pi 4 and a NVIDIA Jetson Nano TX2.

x86 Extension	Microarchitectures	Activation Sparsity	Kernel Sparsity	Sparse Quantization
AMD AVX2	Zen 2, Zen 3	not supported	optimized	emulated
Intel AVX2	Haswell, Broadwell, and newer	not supported	optimized	emulated
Intel AVX-512	Skylake, Cannon Lake, and newer	optimized	optimized	emulated
Intel AVX-512 VNNI (DL Boost)	Cascade Lake, Ice Lake, Cooper Lake, Tiger Lake	optimized	optimized	optimized

Table 2: DeepSparse CPU Hardware Support

3.2 Planning and Implementation Phase

After we had decided which technologies we were going to use for experiment conduction, we started designing the actual architecture of the pipeline and other software around the experiments. The result of this process can be seen in the system diagram depicted in

[3]https://sigopt.com/
[4]https://doku.lrz.de/display/PUBLIC/LRZ+AI+Systems
[5]https://www.lrz.de/
[6]https://www.nvidia.com/en-gb/data-center/tesla-v100/
[7]https://cloud.google.com/compute

Figure 7. Our architecture consists of four main modules (Model Optimization, Benchmarking, Metric Collection and Hyperparameter Tuning). In the following subsections, each module will be explained, and its implementation is briefly discussed.

Figure 7: Overview of our System Architecture

3.2.1 Model Modification

In order to optimize a model with methods such as quantization or sparsification we use the SparseML Toolkit. SpaseML is usually integrated into an existing model training workflow - in our case, a PyTorch[8] implementation for training a deep learning model. The *Train&Modify* component forms the core of the model optimization process. The Experiment Controller consumes a *Base Recipe* and updates the hyperparemters given in *Configuration File*. The result is given to *Train&Modify* to train and export the optimized model. After each run, the transformed model is exported in the Open Neural Network Exchange (ONNX)[9] format and saved together with the used recipe. The entire process and its required dependencies are combined in a single enroot[10] container. Using the SLURM[11] resource manager, the container can then be executed on a specific resource of the LRZ AI Systems.

We have used ResNet-50 [10] for all experiments, as it is widely used in the sparsification literature. For an image classification dataset, CIFAR-100 [14] was chosen as it is a popular benchmark for new approaches. Moreover, it has 50,000 data points, which eliminates a possible computational overhead. We have pretrained a ResNet-50 model on the CIFAR-100 and reached a **baseline accuracy of 80.33%**. The recipe prunes parts

[8] https://pytorch.org/
[9] https://onnx.ai/
[10] https://github.com/NVIDIA/enroot
[11] https://slurm.schedmd.com/

of the convolution weights except in the input convolution and last classification layers by using an Alternating Compressed/ DeCompressed (AC/DC) pruning strategy [17]. Contrary to general magnitude pruning, which prunes part of weights at each iteration, AC/DC divides the training and sparsification process into two phases - decompressed and compressed. During decompressed phase, the full model is trained, and during compressed phase only the Top-k weights are trained. This cycle is repeated several times, and training is completed by final fine-tuning on the sparse model. AC/DC [17] shows not only better models metric wise, but also provides theoretical guarantees of why this method should work.

3.2.2 Benchmarking

The second component of the architecture is the Benchmarking module. It takes the models produced by the Model Optimization module and benchmarks them on the reserved Google Cloud hardware extension as well as on the Edge devices we got. In order to understand the impact of different inference parameters, we tried different possible configurations on the DeepSparse engine mainly multiple batch sizes, core counts for each experiments described in Chapter 4.

3.2.3 Metric Collection

In order to aggregate the different experiment results and configurations for later analysis, we used the Metric Collection component. Metrics from both the training and benchmarking are reported to Weights & Biases. We log the metrics in two different stages (training and benchmarking) under the same run. This gives us the possibility to append the inference results from different hardwares to the corresponding optimized model logs. The metrics we collected are listed below:

- Training metrics: For the training, we keep track of the train and validation accuracy, the loss as well as the model sparsity. Further, we store additional information about the hyperparameters (e.g. epochs, sparsity percentage, architecture, dataset, etc.) used for each run.

- System resources: We collected over 20 different system metrics thanks to the built-in Weights & Biases system monitoring. Along those were metrics like GPU utilization, GPU temperature, memory and disk allocation as well as CPU utilization, just to name a few.

- Inference metrics: For the benchmarking of a sparsified model, we report the latency with respect to different batch sizes and number of cores.

3.2.4 Optimization

As mentioned in 3.1, the SparseML library uses recipes with many hyperparameters to describe the modifications to be applied. To perform hyperparameter optimization, we used the *Configuration Generator* to create multiple configuration files. It takes multiple parameters with corresponding values (e.g. final_sparsity=[0.35,0.75,0.95]) and creates configuration files consisting of all possible combinations between the provided parameter

ranges. We use SigOpt as a parallel approach to find the optimal set of parameters. Finding the optimal set of parameters can be expressed as an optimization problem. We pass the ranges of the parameters, the budget of runs and the formulated optimization problem (e.g. maximize the validation accuracy given the sparsity target) to find the best solution. SigOpt is able to do multi metric optimization (e.g. minimize one metric while maximizing another). Therefore, SigOpt practically replaces the manual configuration generation and creates new parameter values for an experiment. The results of each of the experiments are then reported back to SigOpt, which are then used to derive a new, more optimal set of values. This feedback-refinement loop is repeated for a predefined amount of experiments. The SigOpt UI displays the progress and results of the experiments, including best value combinations and further metrics.

4 Experiments

To define the framework of the experiments, we used the following research questions to guide the experiments.

- How does the sparsity level affect the latency and validation accuracy?
- How does number of cores and batch size affect the final model's latency?
- Do the results obtained above transfer when input size (hence amount of computation) is increased?
- How much does quantization affect the latency?
- What is the effect on train and inference metrics when the base recipes hyperparameters are changed for a fixed sparsification level?
- How do our models perform on the acquired edge devices?
- How is the hardware resource utilization affected by the chosen method and hyperparameter configuration?

This resulted in a total of 4 runs with different objectives. The set of runs is as following:

Grid search run: 54 experimental runs to study the effect of hyperparameter changes to the base recipe. For this we used the configuration generator with the following values (epoch = [10,15,25], pruning_update_frequency = [1,3,5], pruning_epoch_frac=[0.925,0.7,0.5], sparsity in [0.95,0.99]). Our configuration generator create all 54 configuration files, each containing one possible combination of the before mentioned values.

Optimization run: In order to test SigOpts multimetric optimization, we give SigOpt a budget of 20 runs to find the best combinations of hyperparameters to maximize the validation accuracy while minimizing the number of epochs.

Sparsity run: In this run, we vary only the sparsity to see its effect on the validation accuracy and latency. The aim is to find the range where we have the best trade-off between sparsity and accuracy. Therefore, we created 18 different runs with varying sparsities. The first 14 runs test sparsity levels starting from 30% up till 95%, incrementing by 5% each run. The remaining 4 runs test the high sparsity regimes of 96-99%.

Quantization run: In order to check the effect of quantization on our models, we additionally quantized models with 85% and 95% sparsity for later comparison with their non-quantized counterparts.

Apart from these sparsification and quantization experiments, we were also able to experiment on the third methods for reducing the peak memory consumption by operator reordering as discussed in Chapter 2.

5 Results

The following sections show the results of our experiments, which were created using the data and metrics collected in Weights & Biases. Each of the following sections addresses one of the previously presented questions.

5.1 Effect of Sparsity Level on Accuracy and Inference Speed

Varying the sparsity levels while keeping all other configurations the same (shown in Figure 8) shows that at a mid-sparsity level between 40% and up to 95%, we can achieve a validation accuracy higher than the baseline accuracy of 80.33%. The validation accuracy vs sparsity trend follows the Occam's hill displayed in Figure 4. We have used batch

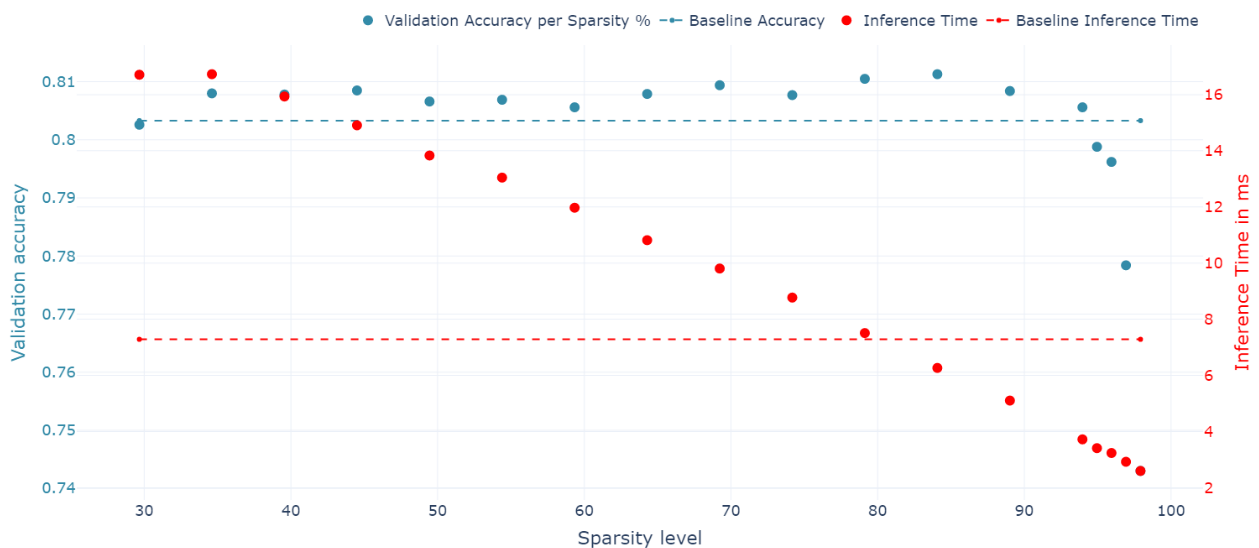


Figure 8: Effect of Sparsity on Inference Speed

size of 1 and 4 cores to benchmark the model at different sparsities. We can see how sparsity affects inference speed. Models with up to 80% sparsity are actually slower than the original non-pruned model. We can get actual speed up only after 80% sparsity. This outcome is caused by DeepSparse engine, as it is optimized towards high sparsity regimes.

5.2 Effect of Batch Size and Number of Cores on Inference Speed

We have investigated the connection of batch size, number of cores and latency (Figure 9). As edge devices have limited resources, we limited the number of cores by 8 and batch size by 64. A 95% sparse model was used for this experiment. Although results go on par with general intuition that increasing number of cores should decrease latency, we see diminishing increase when going from 4 cores to 8. Another curious finding is the highly slight difference of latency for batch sizes of 8 and 16. This effect is seen on any number of cores.

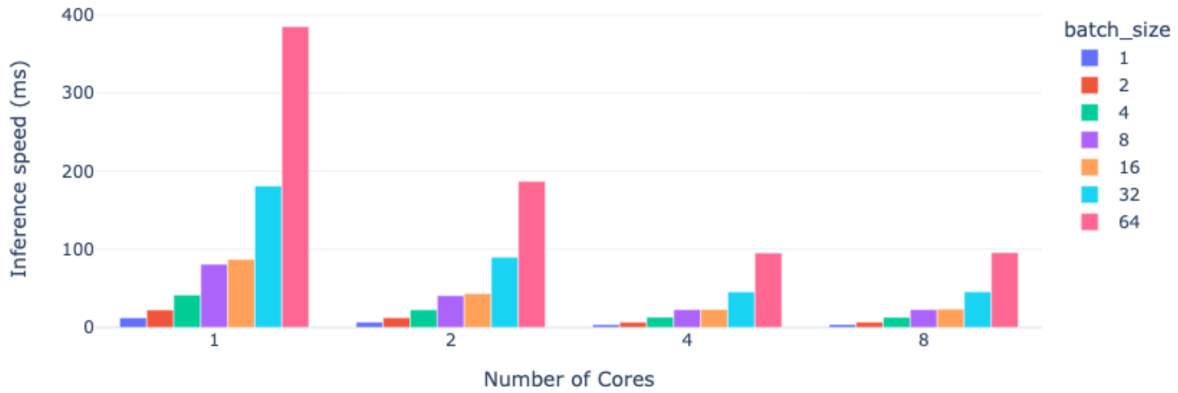


Figure 9: Effect of Batch Size and Number of Cores on Inference Speed

5.3 Effect of the Input Size on Inference Speed

We benchmark the 95% sparse model on various input sizes - 32x32, 128x128, 512x512 - for batch size of 1, 16 and number of cores 1, 2, 4, 8. Figures 10 and 11 show the results for batch size of 1 and 16 respectively (numbers on the bars indicate speedup in percentage relative to dense model).

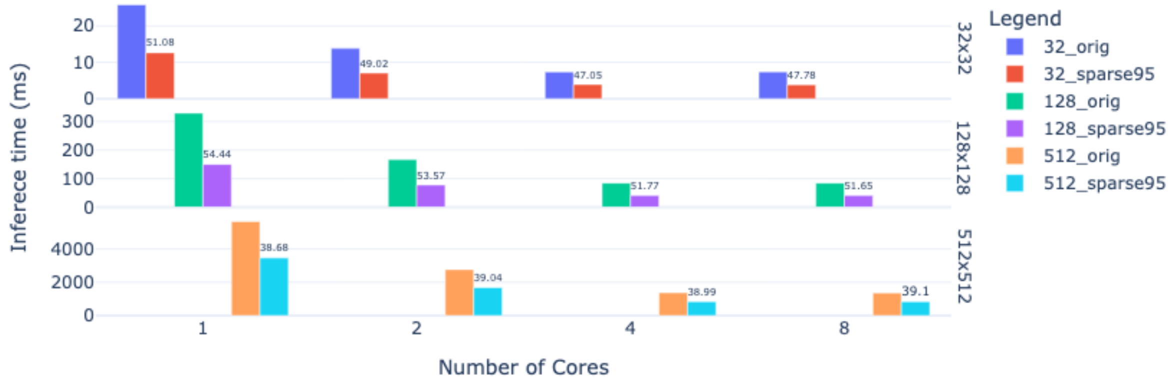


Figure 10: Input Size and Latency (Batch Size 1)

In Figure 10, we can see that increasing the number of cores does not result in a significant speedup relative to the original model. For an input size of 512, the difference is in the range of 1.3%. In absolute terms, a sparse model is 38.68% - 54.44% faster. When switching to batch size 16, we see an even smaller difference in relative latency when increasing the number of cores. Here we can notice that the difference in absolute percentages have increased. For batch size of 16, it ranges from 68.51% to 71.87%. Furthermore, the variance in absolute speedup has significantly decreased.

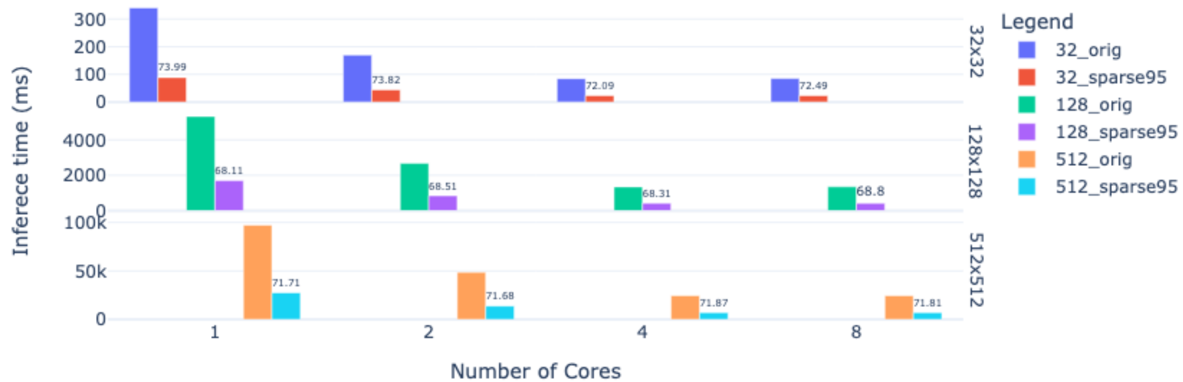


Figure 11: Input Size and Latency (Batch Size 16)

We can conclude that DeepSparse engine handles the increment of batch size much better than of input size.

5.4 Effect of Quantization on Inference Speed

SparseML supports quantization-aware training by adding QuantizationModifier to the recipe. It will perform quantization-aware training during the last several epochs. We can see the effect of quantization on inference speed for different sparsification regimes for batch size 1 and 16 in Figures 12 and 13. The numbers on the bars are the speedup of each model relative to the original model.

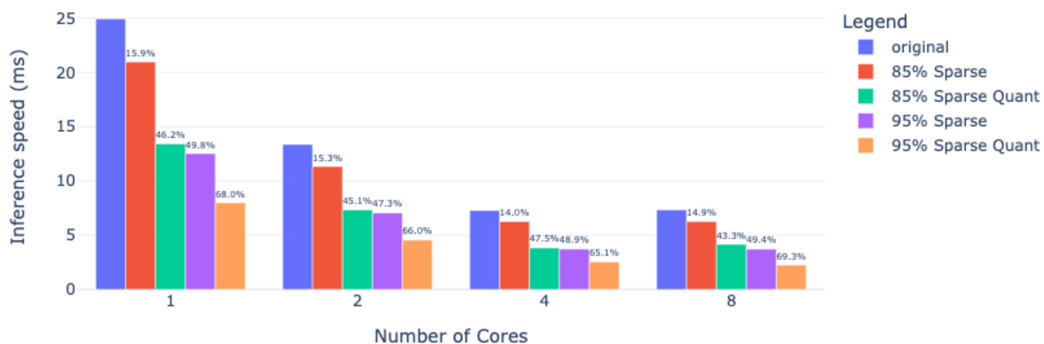


Figure 12: Quantization Effect (Batch Size 1)

We are seeing 20-30% improvement in inference speed when a sparse model is quantized. We can also see that for batch size 1 85% sparsity decreases the inference speed by 14-15% which is not the speedup one expects when pruning 85% of the weights. Furthermore, 85% sparse quantized and 95% sparse model have almost equal inference speed. The reason

can be the way DeepSparse engine handles the pruned weights, and it is more optimized for higher sparsity, such as 95%. The picture changes when we consider batch size 16. We can see almost linear increase in latency of original model, but sparse and quantized models' latency increase 6-8 times. Moreover, the effect of quantization changes for batch size 16. We see around 2 times less speedup compared to batch size of 1.

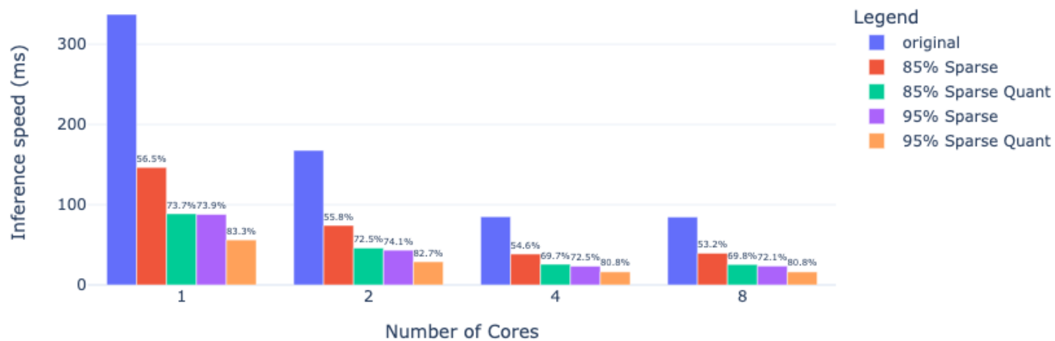


Figure 13: Quantization Effect (Batch Size 16)

5.5 Effect of Recipe Hyperparameters on Train Metrics

To answer this question, we conducted two different types of experiments. First, we collected metrics for 54 different experiments while varying the four main parameters of our base recipe. They consist of the number of epochs and the final sparsity level. Further it includes the pruning_update_frequency, which describes after how many epochs a pruning step should be performed as well as the pruning_epoch_fraction which specifies the percentage of the epochs which should be spend on sparsification besides warm-up and fine-tuning. We tested all combinations which are possible with the values below:

- epoch in [10,15,25]
- pruning_update_frequency in [1,3,5]
- pruning_epoch_frac in [0.925,0.7,0.5]
- sparsity in [0.95,0.99]

In addition, we used SigOpt to perform a multimetric optimization of the parameters with the goal of generating an accurate model in as little time as possible. We instruced SigOpt to derive the best set of parameters within a specific range which minimizes the used amount of epochs while also maximizes the validation accuracy. For this purpose we used the same hyperparameters as above and defined the following ranges

- epoch in (min=10, max=20)
- pruning_update_frequency in (min=1, max=7)

- pruning_epoch_frac in (min=0.5, max=1.0)
- sparsity in [0.95,0.99]

Unfortunately, in both experiments no real correlations between the parameters of pruning_update_frequency and pruning_epoch_frac on the validation accuracy could be found. Besides the trivial conclusion that with lesser epochs and increased sparsity the validation accuracy decreases, almost all combinations reach a similar validation score (the variance is only around 3%). This can be seen in Figure 14, which shows validation accuracy for the different combinations of the pruning fraction and update frequency for the 95% sparsified model. The reason for this could be due to the fact that the underlying method of AC/DC sparsification is not as sensitive to changes in hyperparameter settings or the CIFAR-100 task is still to trivial to see real differences. However, one interesting thing to observe is SigOpts multimetric optimization in Figure 15, which runs a lot more experiments with smaller epoch numbers due to its minimization obejctive.

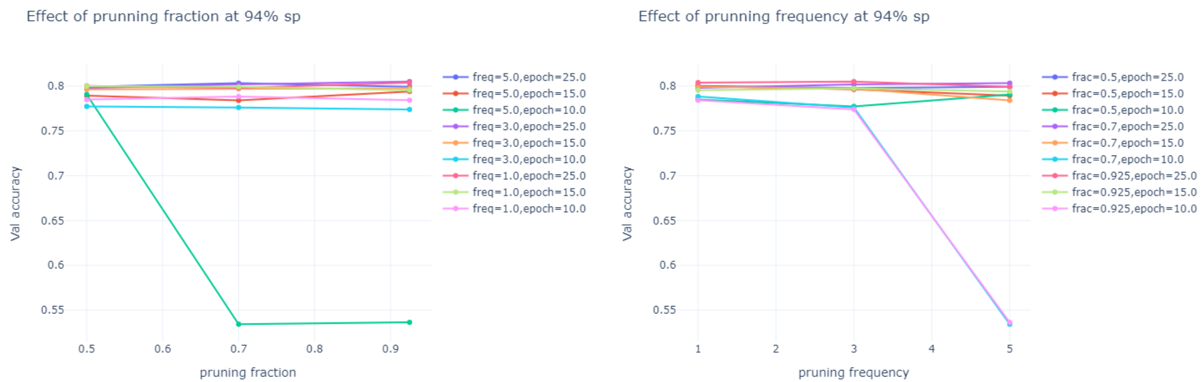


Figure 14: Pruning Fraction and Pruning Frequency Effect on the Validation Accuracy

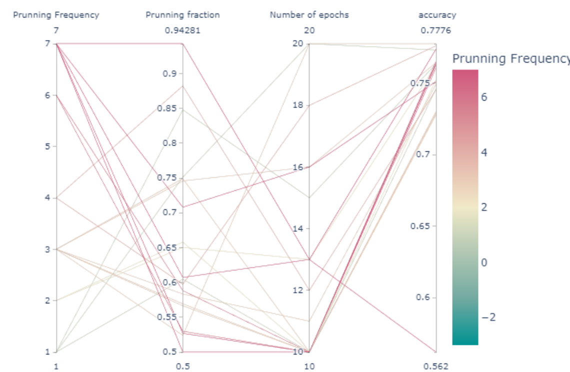


Figure 15: SigOpt Parallel Coordinates of Recipe Parameters on Validation Accuracy

5.6 Model Deployment on different Edge Devices

In order to test our results in a real scenario we used some of our own devices. We were able to deploy the above said models and experiments on the real-time edge devices - Raspberry Pi 3, Raspberry Pi 3 B+, Raspberry Pi 4 and NVIDIA Jetson Nano TX2. From table 2 it is clear that DeepSparse doesn't support ARM based architectures. So, we have exported the model files and benchmarked the models and edge devices using the native libraries available. The table 3 will showcase the specifications of the edge devices chosen.

Device	CPU	Cores	CPU Clock	GPU	RAM
Raspberry Pi 3	Cortex-A53 64-bit	4	1.2 GHz	VideoCore IV	1 GB DDR2
Raspberry Pi 3 B+	Cortex-A53 64-bit	4	1.4 GHz	VideoCore IV	1 GB DDR2
Raspberry Pi 4	Cortex-A72 64-bit	4	1.5 GHz	VideoCore VI	4 GB DDR4
NVIDIA Jetson TX2	Cortex-A72 64-bit	4	1.2 GHz	Pascal GPU with 256 CUDA cores	4 GB DDR4

Table 3: Used Edge Devices Specifications

The edge devices specified in the table above are chosen for the models to be deployed and benchmarked. We chose the baseline model, 85% sparsified, 85% sparsified+quantized, 95% sparsified model and 95% sparsified+quantized models for deployment across the said edge devices. It is observed that the models took a little more inference time on the edge devices when compared to the DeepSparse benchmarking engine. The fact that Raspberry Pi 3 and 3 B+ models' specification are more or less same is reflected in the inferencing time of the models. Also the Jetson TX2 CPU is similar to Raspberry Pi 4's performance. Figure 16 gives an in detail comparison across models and edge devices.

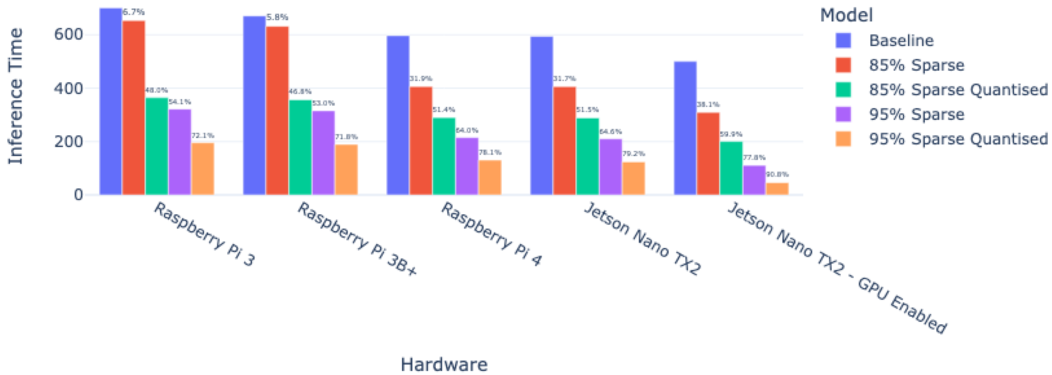


Figure 16: Comparison of Models Deployed on Edge Device

From the graph it is observed that Jetson TX2 with GPU enabled performed better across the models and the 95% sparsified+quantized model is 10X faster when compared to the baseline model in terms of inference time. When we look at the relationship across the models on the edge device, we can clearly see it follows a pattern of reduction in inference time with better hardware configuration. The trend is similar to the hardware listed in table 3. Apart from this, it is also observed, it holds general relationship as analogous to the DeepSparse benchmarked models.

Using the reordering of operators technique discussed in 2.3, we have reordered the operators for the baseline model. We were able to see that there was a 10% reduction in the model's memory footprint per inference when compared to the baseline model on the edge devices. The memory footprint of the models were same on all the edge devices. It is seen that baseline required 300 kilobytes of memory for inferencing, where as after the reordering operation, it required only 272 kilobytes, which is approx. 10% reduction in memory size. Figure 17 visualizes the differences in the memory footprint.

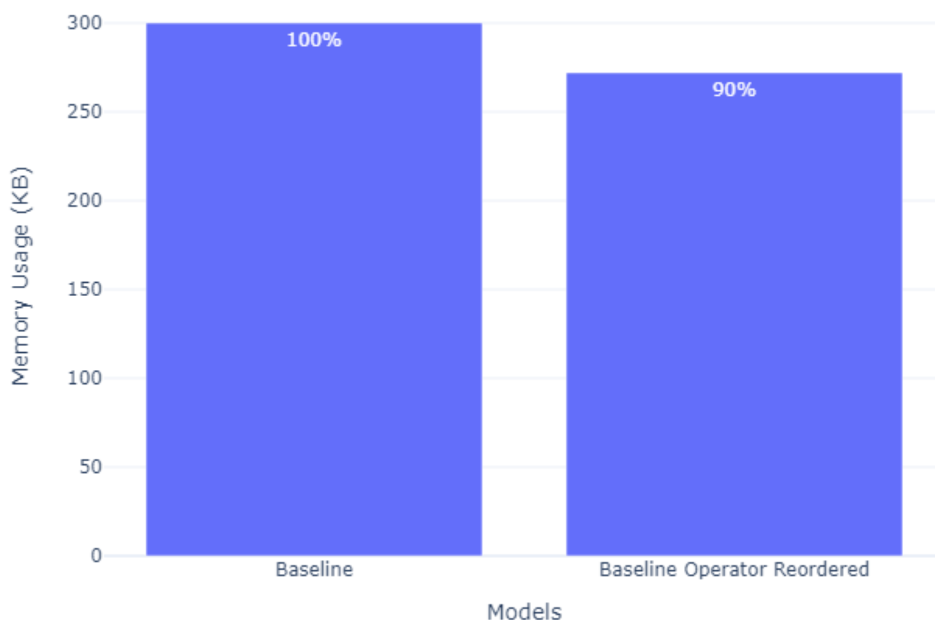


Figure 17: Comparison of Models' Memory Footprint on Edge Devices

5.7 System Resources Usage

During the process of modifying the models, we kept track on the system resources consumption over time and compared their allocation and performance. We observed that the resources utilization is independent of any hyperparameter choice, like sparsity or pruning frequency. The system resources allocation always follows the same pattern. The only correlation is the obvious one between number of epochs and training time.

The process of sparsifying or quantizing the model is computationally intensive as it requires retraining of the models for multiple epochs. Therefore, a typical run (sparsifying a model in 20 epochs) took around 30 minutes to complete on the before described LRZ GPUs. The GPUs used around 200 Watts (as shown in figure 18 left) and 100% of the GPU cores throughout the sparsification process. An interesting thing to observe

is that the quantization process was less computationally intensive than the sparsification. One could see that the GPU power consumption as well as utilization dropped to around 150 Watts and 70% utilization as soon as quantization started (when the yellow line drops).

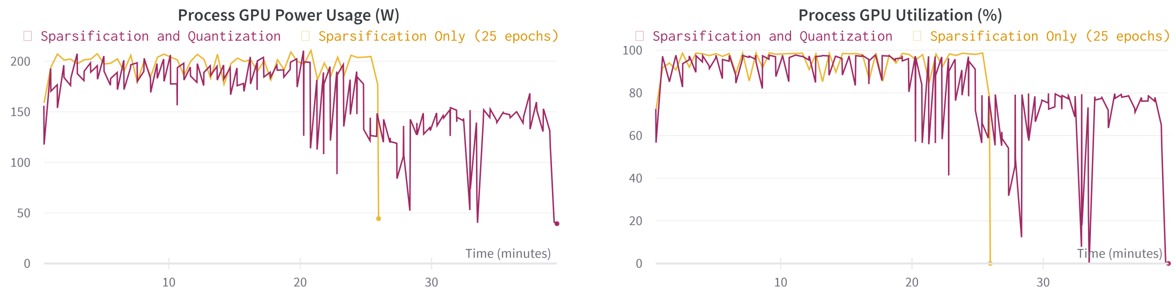


Figure 18: GPU Power Usage (left)/ GPU Memory Usage (right) for only Sparsification and Sparsification with Subsequent Quantization

6 Conclusion and Further Research

The limited resources of edge devices provide unique challenges to researches in this field. And even though the huge benefits of utilizing such devices on a large scale heavily outweigh these challenges, a lot of research still has to be conducted.

Therefore, this work conducted a literature review to identify and describe the most promising techniques of the current state-of-the-art approaches and hardware limitations. Especially sparsification and quantization have been identified as very promising methods. To demonstrate how those techniques can be used and optimized, a prototype of a deep learning pipeline was designed and implemented. This pipeline allows to efficiently tune and test different sparsification and quantization methods without too much technical expertise. In the course of the project, the pipeline was used to make a set of experiments to further strengthen the understanding of how train and inference metrics, input sizes, batch sizes and CPU cores, as well as sparsification, model size, and hyperparameter are connected and should be interpreted.

Using our trained ResNet-50 architecture on the CIFAR-100 dataset, we have shown that we can achieve higher accuracy and lower runtime than the baseline in a high sparsity regime (more than 85% sparsity). We showed that increasing the number of cores from 1 core up to 4 cores showed a significant increase in the speed of deployment, while levering to 8 cores didn't show significant results compared to using only 4 cores. The experimentation of different combinations of input sizes, batch sizes and number of cores showed that we can get higher speed up compared to the initial model, when increasing the batch size. Our results indicate that increasing number of cores do not affect the relative speedup compared to original model. Combining sparsification with quantization gave a huge speedup of 65-82% in latency.

We have also deployed the above discussed models on real-time edge devices like Raspberry Pi 3, Pi 3B+, Pi4 and NVIDIA Jetson Nano TX2 and compared the inference times. It is observed that among the devices deployed, Jetson Nano TX2 had lesser inference time and the fact lies with the specification of the device, as it has a GPU with 256 CUDA cores.

In summary, we have shown that there are promising opportunities to deploy deep learning models on edge devices. However, one inevitably has to decide between different trade-offs. The faster the model, the lower the accuracy and the faster and more powerful the edge device the more expensive the resource. In the future, there will be a lot of promising research in different areas around this domains. Efficient, cheap and powerful hardware devices need to be created to deploy efficient models. Furthermore, research will continue to reduce the size of models faster and more efficiently. Another interesting point of research is the development of one-shot methods to reduce the size of models so that the additional effort of fine-tuning and retraining is eliminated. With the popularity of giant language models that consume vast sums of money during training, the developments in the field of sparse training are also extremely exciting.

References

[1] Tianlong Chen et al. "The Lottery Ticket Hypothesis for Pre-trained BERT Networks". In: *CoRR* abs/2007.12223 (2020). arXiv: 2007.12223. URL: https://arxiv.org/abs/2007.12223.

[2] Tianlong Chen et al. "The Lottery Tickets Hypothesis for Supervised and Self-supervised Pre-training in Computer Vision Models". In: *CoRR* abs/2012.06908 (2020). arXiv: 2012.06908. URL: https://arxiv.org/abs/2012.06908.

[3] *Cisco visual networking index: Global mobile data traffic forecast update (2017â"2022).* online: http://www.cisco.com/c/en/us/solutions/collateral/service-provider/visual-networking-index-vni/mobile-white-paper-c11-520862.html. last accessed: 18 July 2022.

[4] Tim Dettmers and Luke Zettlemoyer. *Sparse Networks from Scratch: Faster Training without Losing Performance.* 2019. DOI: 10.48550/ARXIV.1907.04840. URL: https://arxiv.org/abs/1907.04840.

[5] Utku Evci et al. "The Difficulty of Training Sparse Neural Networks". In: *CoRR* abs/1906.10732 (2019). arXiv: 1906.10732. URL: http://arxiv.org/abs/1906.10732.

[6] Jonathan Frankle and Michael Carbin. "The Lottery Ticket Hypothesis: Finding Sparse, Trainable Neural Networks." In: *ICLR*. OpenReview.net, 2019. URL: http://dblp.uni-trier.de/db/conf/iclr/iclr2019.html#FrankleC19.

[7] Trevor Gale, Erich Elsen, and Sara Hooker. *The State of Sparsity in Deep Neural Networks.* 2019. DOI: 10.48550/ARXIV.1902.09574. URL: https://arxiv.org/abs/1902.09574.

[8] Amir Gholami et al. "A survey of quantization methods for efficient neural network inference". In: *arXiv preprint arXiv:2103.13630* (2021).

[9] Joydeep Ghosh and Kagan Tumer. "Structural adaptation and generalization in supervised feed-forward networks, d". In: *Artif. Neural Networks* (1994), p. 458.

[10] Kaiming He et al. "Deep residual learning for image recognition". In: *Proceedings of the IEEE conference on computer vision and pattern recognition.* 2016, pp. 770–778.

[11] Torsten Hoefler et al. "Sparsity in Deep Learning: Pruning and growth for efficient inference and training in neural networks". In: *CoRR* abs/2102.00554 (2021). arXiv: 2102.00554. URL: https://arxiv.org/abs/2102.00554.

[12] *IoT: Number of Connected Devices Worldwide 2012â"2025.* online: https://www.statista.com/statistics/471264/iot-number-of-connected-devices-worldwide/. last accessed: 18 July 2022.

[13] Xiaojie Jin et al. "Training Skinny Deep Neural Networks with Iterative Hard Thresholding Methods". In: *CoRR* abs/1607.05423 (2016). arXiv: 1607.05423. URL: http://arxiv.org/abs/1607.05423.

[14] Alex Krizhevsky, Geoffrey Hinton, et al. "Learning multiple layers of features from tiny images". In: (2009).

[15] Edgar Liberis and Nicholas D. Lane. *Neural networks on microcontrollers: saving memory at inference via operator reordering*. 2019. DOI: 10.48550/ARXIV.1910.05110. URL: https://arxiv.org/abs/1910.05110.

[16] Massimo Merenda, Carlo Porcaro, and Demetrio Iero. "Edge Machine Learning for AI-Enabled IoT Devices: A Review". In: *Sensors* 20.9 (Apr. 2020), p. 2533. DOI: 10.3390/s20092533.

[17] Alexandra Peste et al. "Ac/dc: Alternating compressed/decompressed training of deep neural networks". In: *Advances in Neural Information Processing Systems* 34 (2021), pp. 8557–8570.

[18] C. E. Rasmussen and Z. Ghahramani. "Occam's razor". In: *Advances in Neural Information Processing Systems 13*. Ed. by T. K. Leen, T. G. Dietterich, and V. Tresp. Cambridge, MA: MIT Press, 2001, pp. 294–300.

[19] Claude Elwood Shannon. "A mathematical theory of communication". In: *The Bell system technical journal* 27.3 (1948), pp. 379–423.

[20] Weisong Shi et al. "Edge Computing: Vision and Challenges". In: *IEEE Internet of Things Journal* 3.5 (2016), pp. 637–646. DOI: 10.1109/JIOT.2016.2579198.

[21] Nitish Srivastava et al. "Dropout: A Simple Way to Prevent Neural Networks from Overfitting". In: *Journal of Machine Learning Research* 15.56 (2014), pp. 1929–1958. URL: http://jmlr.org/papers/v15/srivastava14a.html.

[22] Yi Sun, Xiaogang Wang, and Xiaoou Tang. "Sparsifying Neural Network Connections for Face Recognition". In: *CoRR* abs/1512.01891 (2015). arXiv: 1512.01891. URL: http://arxiv.org/abs/1512.01891.

[23] *Top 5 benefits of edge computing for businesses*. online: https://www.techtarget.com/iotagenda/tip/Top-5-benefits-of-edge-computing-for-businesses. last accessed: 18 July 2022.

[24] Blesson Varghese et al. "Challenges and Opportunities in Edge Computing". In: *2016 IEEE International Conference on Smart Cloud (SmartCloud)*. 2016, pp. 20–26. DOI: 10.1109/SmartCloud.2016.18.

[25] *What Edge Computing Means for Infrastructure and Operations Leaders*. online: https://www.gartner.com/smarterwithgartner/what-edge-computing-means-for-infrastructure-and-operations-leaders. last accessed: 18 July 2022.

[26] Dianlei Xu et al. "A Survey on Edge Intelligence". In: *CoRR* abs/2003.12172 (2020). arXiv: 2003.12172. URL: https://arxiv.org/abs/2003.12172.

[27] Shuochao Yao et al. *DeepIoT: Compressing Deep Neural Network Structures for Sensing Systems with a Compressor-Critic Framework*. 2017. DOI: 10.48550/ARXIV.1706.01215. URL: https://arxiv.org/abs/1706.01215.

[28] Haonan Yu et al. *Playing the lottery with rewards and multiple languages: lottery tickets in RL and NLP*. 2019. DOI: 10.48550/ARXIV.1906.02768. URL: https://arxiv.org/abs/1906.02768.