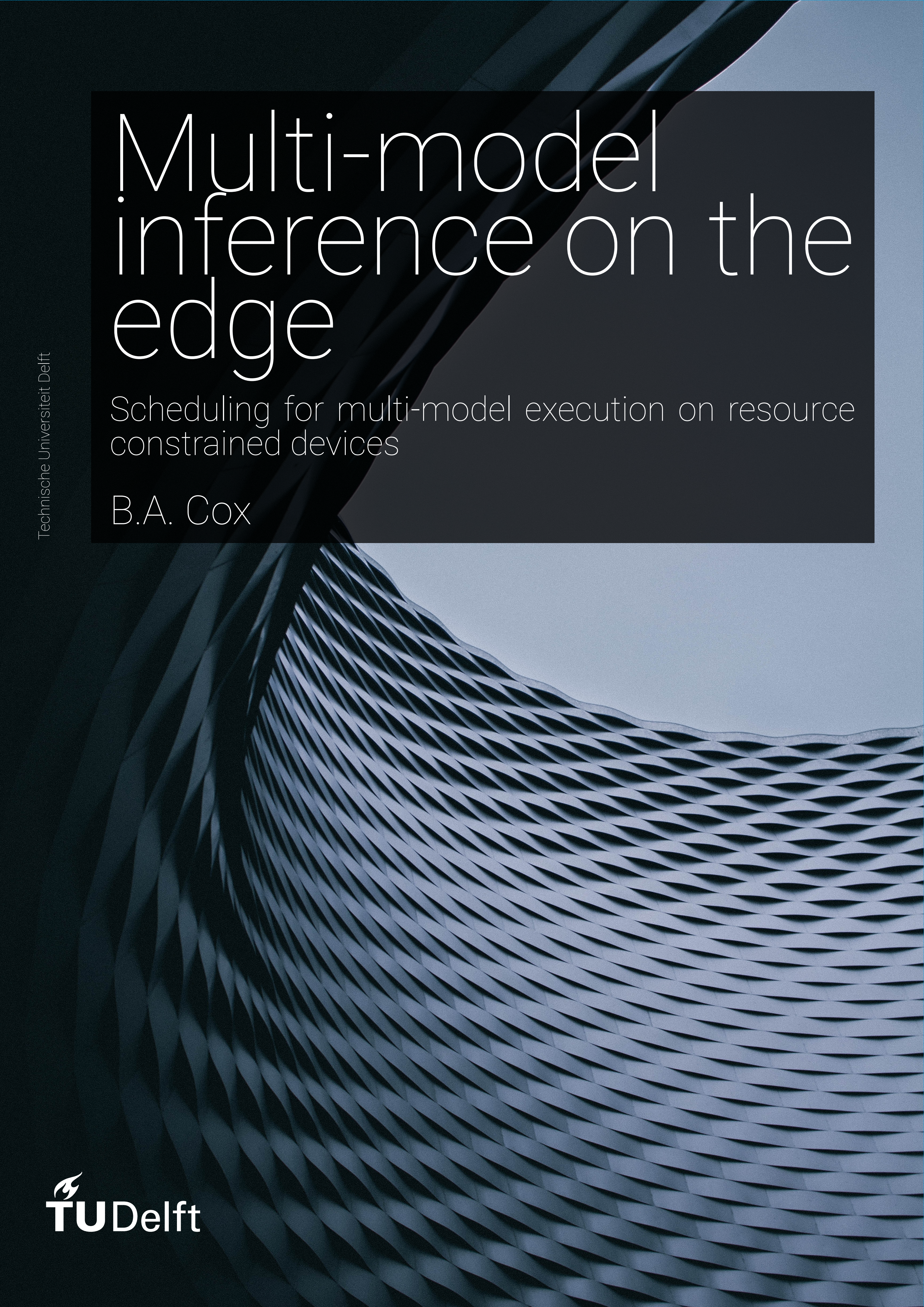
Multi-model inference on the edge

Scheduling for multi-model execution on resource constrained devices

B.A. Cox



Multi-model inference on the edge

Scheduling for multi-model execution on resource constrained devices

by

B.A. Cox

to obtain the degree of Master of Science
in Computer Science, within the field of Data Science & Technology,
specialized in Distributed Systems, at the Delft University of Technology,
to be defended publicly on Monday November 9, 2020 at 11:00 AM.

Student number:	4159470	
Project duration:	February 1, 2019 – November 9, 2020	
Thesis committee:	Dr. L. Chen,	TU Delft, supervisor
	Prof. dr. ir. D.H.J. Epema,	TU Delft
	Dr. M.A. Zuniga,	TU Delft

TU Delft

Abstract

Deep neural networks (DNNs) are becoming the core components of many applications running on edge devices,especially for image-based analysis, e.g., identifying objects, faces, and genders. While very successful in resource rich environments like the cloud of powerful computers, utilizing Deep Learning on edge devices for inference is not used to great extend. The resource constraints of edge devices present bottlenecks that prevent all but small networks from on-device inference. Existing solutions like compression or off-loading to the cloud sacrifice either model accuracy or inference latency.

To counter the aforementioned shortcomings, MASA and EDGECAFFE are proposed as a solution. The process of DNN inference by existing frameworks is to loaded and execute a model in its entirety. As this ignores the resource limitations that are inherent to edge devices, a new framework EDGECAFFE is proposed. EDGECAFFE is able to execute models partially and thereby reducing the memory footprint of the model. Additional, EDGECAFFE allows for coordination between models when a multi-model DNN inference is executed.

MASA, a memory-aware multi-DNN scheduling algorithm, featuring on modeling inter-and intra-network dependency and leveraging complimentary memory usage of each layer. MASA can consistently ensure the average response time when deterministically and stochastically executing multiple DNN-based image analyses.

With extensive evaluations, MASA can consistently ensure the average response time when deterministically and stochastically executing multiple DNN-based image analyses. We extensively evaluate MASA on three configurations of Raspberry Pi and a large set of popular DNN models triggered by different arrival patterns of images. Our evaluation results show that MASA can achieve lower average response times by up to 8× and 16× for deterministic and stochastic arrivals respectively on devices with small memory, i.e., 512 MB to 1 GB, compared to the state of the art multi-DNN scheduling solutions.

Preface

This project started one year ago with a literature survey. Twelve months later this thesis is the result of the continuation of that same work. While being immersed in the world of Edge Computing and Deep Learning, a great many things that I have learned originated from those fields as well as from personal development and critical thinking.
During this project I had the pleasure of working with kind and talented people, some of whom I would like to thank. First and foremost I would like to express my gratitude towards my supervisor Lydia Chen. Without the constant drive and presented opportunities, this project would be less colorful and exciting than it turned out to be. The weekly talks helped to great extend to keep the project on track. I am very grateful for the confidence you had in the project when suggesting this work to be used in the Bachelor Research projects. This all has made it very pleasant and exiting to work on this project.
I would like to thank Amirmasoud Ghiassi and Robert Birke for your interest and contributions to the project. Thanks to Jeroen Galjaard, for contributing to the EDGECAFFE project. It was very pleasant to have an active contributor to the project who was not afraid of changing the existing code.
Finally, I would like to thank Eva, who is always there when needed. Your never-ending support and remarks always keep me on track.

B.A. Cox
Delft, November 2020

Contents

1 Introduction

The rise of edge computing has had an explosive growth the past years. Due to rise of Internet of Things (IoT) and 5G, edge computing is being used more and more. More than 25 billion IoT-devices are estimated to be connected to the Internet [40]. The nature of edge devices makes them resource constrained, either in computational power, communication bandwidth, available energy, or latency. Small cloudlets, embedded IoT-devices, and mobile phones can function as an edge device. This means that edge devices are heterogeneous in terms of hardware. Edge computing moves the computations away from server in the cloud and tries to execute the computations as close to the source as possible. At the same time, Deep Learning is widely used in numerous fields and industries. Combining edge computing and Deep Learning to provide intelligence at the edge is an important step to create ubiquitous application. With Deep Learning predominantly running in large cloud systems and with the mobile connectivity, due to new wireless technologies like 5G, is there a need to run Deep Learning inference on edge devices.

Edge devices, such as cameras, wearables, and sensors, are becoming an integral part of our daily life and increasingly draw on deep neural networks (DNNs) for sophisticated real-time image-based analysis. Instagram [67] enables real-time knowledge extraction on user's devices upon captures of images via running the *inference* of convolutional neural networks (CNNs) – one of the most widely adopted type of DNN for image processing [60, 65]. Autonomous vehicles and surveillance systems are other examples that need execute DNN inferences to recognize the surroundings and facilitate on-line decision making. It is imperative to ensure the responsiveness of DNN inference, i.e., low response time, on edge devices for the quality of experience as well as safety.

Depending on the purposes of the learning tasks, DNNs can have disparate architectures in terms of the number and size of layers. For instance, YOLO [55] can efficiently identify objects, whereas AgeNet and GenderNet [34] can discern the age and gender of people. To extract the numerous information embedded in a single image, multiple different DNN inferences need to be executed – greatly increasing the computational overhead, and the risk of resource contention as well as unresponsiveness. Moreover, images can be taken periodically or stochastically and trigger complex DNNs execution flows, e.g., GenderNet can be trigger after faces are recognized by FaceNet.

We term multi-DNN inference job the analysis of a same image processed by a set of DNNs. The difficulty of maintaining low response times of both periodic and stochastically multi-DNN inference jobs is significantly higher, due to the slowdown dynamics across small and large inference networks.

DNNs are known for the high accuracy of complex tasks at the cost of intensive resource footprint and computation overhead. Typical CNN models [28, 59] can take up 892 MB memory to store 57.7 million model parameters. Executing the convolution layers of CNNs is CPU intensive, whereas computing the fully-connected layers is more memory intensive due to the high number of model weights. To turn the DNN edge inference from infeasible to reality, a large body of related studies [4, 42] aims to minimize the resource demands of DNNs by compressing and pruning models for *individual* networks – achieving a calculable trade-off with the accuracy. As for multi-DNN inferences, the existing DNN frameworks, e.g., PyTorch [50], and Caffe [23], support only sequential execution.

1.1. Problem definition and research questions

Deep Neural Networks (DNN) mostly are trained in the cloud. These limitless environments in terms of resources tend to create larger and larger models. To use these trained models, they often are deployed on Edge devices. Edge devices, limited in resources, not always equipped to execute trained models for inference. A single models often causes problems for an Edge devices, multiple models are very rarely executed on Edge devices simultaneously. The need for Edge inference and multi-model Edge inference becomes more emergent with increase of the number of edge devices.

Problem Definition: How can multi-model DNN inference be facilitated in a resource constrained environment?

RQ1. *How can DNNs be decomposed in smaller components?*

RQ2. *How can the execution of a DNN be speed up using the multiple CPU cores?*

RQ3. *What are the resource needs of a DNN?*

RQ4. *What scheduling algorithm can be used to schedule multiple DNNs on resource constrained devices?*

RQ5. *How can multi-model DNN inference scale on different devices with different resources?*

1.2. Contributions

The contributions of this research is four-fold. First of al, we present a deterministic model that is able the schedule multiple DNNs while respecting the resource limitations. Secondly, we present a new DNN inference framework called EDGECAFFE. This framework allows for the partial (or layer-to-layer) execution of DNN layers. Fine-grained control over the resources is ensured by EDGECAFFE. The lack of these features in existing frameworks shows the need for EDGECAFFE when multi-model DNN inference on the edge become reality. Thirdly, using the fine-grained control of the execution of DNN models, we developed a resource aware scheduling algorithm MASA that aims to schedule multi-model DNNs in resource constrained environments. Lastly, with extensive experimentation we show the effects of multi-model DNN inference on resource constraints devices.

1.3. Organization

This report is divided in five parts. chapter 2 presents the foundation of background knowledge that is required for this problem. This is continued in chapter 3 where the recent developments are explained in the various related fields of research. The second part of this document can be found in chapter 4 where the inner workings and properties of Deep Neural Networks are discussed. This chapter continues by explaining how the structure to DNNs can be used to accelerate the execution during inference. chapter 5 provides a new framework that allows for the partial execution DNNs, which is an intricate part of the solution. With the tools in place, chapter 6 proposes a new algorithm that combats the limitations from current solutions. Extensive evaluation of the proposed solutions is shown in chapter 7. Finally in chapter 8, the key findings and future work is discussed.

2 Background

Edge Inference is a the intersection of the fields Deep Learning and Edge Computing. In this chapter we discuss background information that is related to Edge Inference. We start in section 2.1 by looking into Deep Learning. Secondly, Edge Inference as a subfield of Edge Computing is discussed in section 2.2.

2.1. Deep Learning

Deep Learning (DL) is a branch of machine learning, where features are learned from a (large) set of examples. The automated manner in which Deep Learning can extract and learn features from raw data is an advantage over classical machine learning. Deep Learning leverages an Artificial Neural Network (ANN) to learn patterns and distributions in data. The network is composed in a layer-wise structure where the nodes in layer N is connected with the nodes in layer $N+1$. The network consists of an input layer followed by hidden layers and an output layer (figure 2.1). An Deep Neural Network is an ANN multiple hidden layers. A DNN can be trained by minimizing the loss function using the gradient descent. Using back-propagation the errors in the model can be corrected and help the model converge over time. A trained model can be used for inference using forward-propagation.

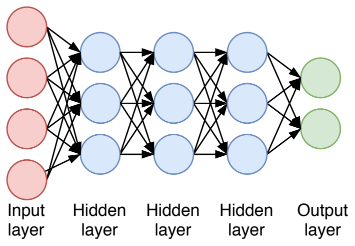


Figure 2.1: Structure of an ANN

2.1.1. Deep Neural Networks

A deep learning is used to solve a multitude of problems, different networks with different specialties exist. The most known networks are discussed in this part of the chapter.

Fully Connected Neural Network (FCNN)

The output of each layers is forwarded to each node in the next layer. Because each node is connected to all nodes of the next layer, when the number layers or the number of nodes per layer increases, the number of parameters in the model increases dramatically. FCNN can be used for function approximation and extraction of features. When the complexity of the problem rises, the FCNN is slow to converge and slow in performance.

Convolutional Neural Networks

Convolutional Neural Networks (CNN) are extensively used in the field of image classification. A CNN gets a 2D input and extracts high level features. The layers of a CNN consists of convolutional layers, pooling layers, and fully connected layers. The convolutional layers have filters (also called kernels) that creates a feature map based on the input. The pooling layers reduce the spatial size of the feature map. At the end of the networks, the fully connected layers are placed to classify the input the into various categories. CNNs are one of the main types of DNNs for image-based inference. They combine convolution and pooling layers for feature extraction and complementary fully-connected layers for classification [28, 32, 59]. Figure 2.2 presents an exemplary architecture. CNNs are inspired by the organization of the visual cortex. Individual neurons answer to stimuli only in a restricted region of the visual field named receptive field. Multiple such fields overlap to cover the entire visual area. Convolutional layers divide the image in receptive fields from which they extract features via convolution with a filter matrix that is shared across receptive fields. Initial layers capture low-level features, e.g., edges and color. Subsequent layers combine low-level characteristics into higher-level features which allow the network to gain a thorough understanding of the images. Pooling layers apply aggregation functions, e.g. `max`, on fields of convolved features. This reduces noise and extracts dominant features which are positional and rotational invariant. Fully-connected layers learn non-linear combinations of the extracted high-level features and classify the image using the `softmax` classification technique. Different CNN architectures vary the number and hyper-parameters of these layers.

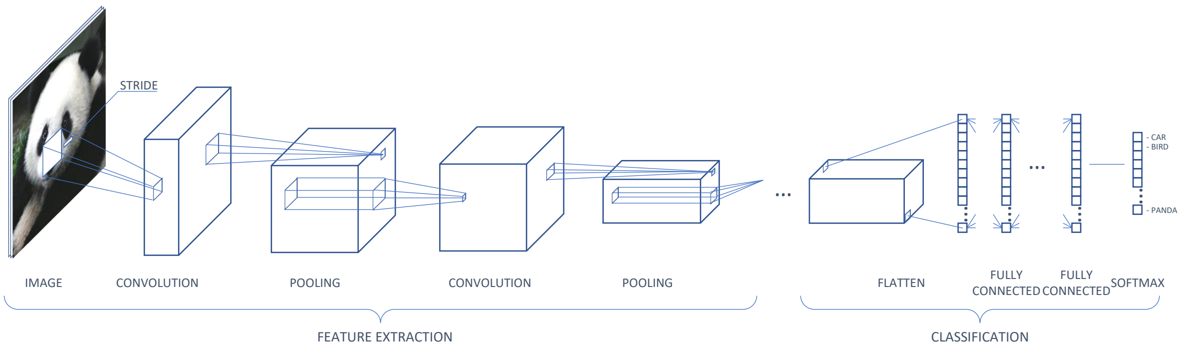


Figure 2.2: CNN architecture

Recurrent Neural Networks

Recurrent Neural Networks (RNN) are designed to handle sequential data. While linear DNNs like CNNs use a single data point as input, uses a RNN a sequential set of data points as input. This makes RNNs suitable for inference on sequential data like time series. A problem that can occur by training RNNs is gradient explosion or gradient vanishing. To overcome this problem the Long short-term memory (LSTM) network has proven to mitigate these problems. LSTMs have a more complex memory cell, than RNNs, with gates that control the flow of information to the memory cell. The recurrent nature of an RNN makes it more difficult to model it as a network graph. RNNs can be unrolled to remove the recurrent relationship in the memory cells. Unrolled RNNs are processed iterative, as the state of the current node is dependent on the state of the previous node.

Generative Adversarial Networks

A Generative Adversarial Networks (GAN) consist of a generative network and a discriminator network. The goal of the generative network is the learn the data distribution of the training data while the discriminator network needs to correctly determine if the given data is coming from the true data or is generated by the generative network. GANs are often used for generative tasks. A GAN can be used for inference tasks by using either the discriminator of the generator network.

Auto-Encoders (AE)

AE consists often out of two neural networks. The first part tries to learn the features of the input data, while the second part tries to restore the input-data from the learned features. The structure of an AE consists of high dimensional input and output layers with a smaller dimensional hidden layer acting a bottleneck. AE are often used to classify data with a high dimensionality of features.

Deep Reinforcement Learning (DRL)

Deep reinforcement learning combines ANNs with reinforcement learning. DRL tries to learn a policy that maps the state of the agent to a set of actions. DRL uses feedback from the environment to train ANN that represents the policy (figure 2.3). After training, the ANN can be used during inference to select the best action based on the state. DRL trains a DNN that maps the given state to an action. This trained model can be used for inference by a forward propagation true through the model.

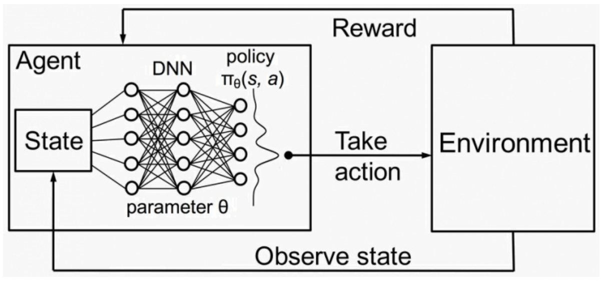


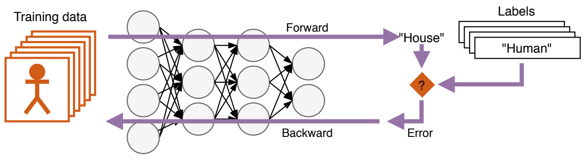
Figure 2.3: Deep reinforcement learning[1]

2.1.2. Training and Inference

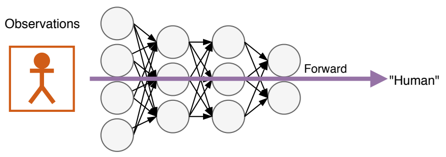
In order to used a DNN, first it needs to be trained. This is done in three steps: the forward pass, calculate the loss, and the backward pass. This process can be seen in Figure 2.4a. In the training phase input data is given to the network. A forward pass is done where for each layer and each neuron, the output is calculated where W are the weights and b the bias in a neuron. The output of a single layer is used as the input of the next layer. The results of these cascading transformations on the data are the values in the output layer.

$$y = Wx + b \tag{2.1}$$

Since the true labels of the data are known during training we can correct the network when mistake are made. The error made by the network is determined by the loss function. The cross-entropy loss is a well used loss function to determine the loss between the output of the forward pass and the true label. In case a regression task is trained on a network the mean squared loss (L_2 loss) is used to calculate the loss. To correct the network for the loss, the well known backpropagation algorithm is used. For each neuron, the gradient is calculated to adjust the weights for the loss. Optimizers such as Stochastic Gradient Descent (SGD), Mini-batch Gradient Descent, and Adaptive Moment Estimation (Adam) can be used to improve the training performance. When the accuracy of the network has converged enough, hence the difference between consecutive loss values is smaller than a certain threshold.



(a) Training a neural network



(b) Using a trained neural network for inference

Figure 2.4: Deep Learning training and inference

In the inference phase, a trained network can be deployed and used for inference. In this case only the forward pass (Figure 2.4b), described above, is executed. The network is given unseen data and is asked to make a prediction based on what is has learned. The prediction can be either a classification or a regression outcome, depending on the learned task. The outcome of the network is not corrected and can be used for other applications.

2.2. Edge Inference

Edge Inference is a subfield within Edge Computing. This subfield looks for efficient methods of allowing DNNs to run on edge devices for the purpose of inference. The target hardware in that case (edge devices) have a distinction from devices that live in the cloud. Edge devices have a limited set of resources that can not (or with great difficulty) be changed. This can be the result of the location where the edge devices are

[1]`http://amid.fish/reproducing-deep-rl`

located or the form factor. Edge devices can be deployed in the field, as is done with Internet Of Things (IoT) devices, which make the hardware logistically difficult to access. Hardware in devices like mobile phones are generally difficult to upgrade due to the compact form factor.

Edge devices are located at the edge of the network, close to the source of the data and close to the users. The resource constrained nature of the edge devices makes running Deep Learning inference models difficult. The inference models are trained on large clusters in the cloud on huge amounts of data, to make the model accurate and general enough to use in the real world. The use of practical unlimited resources in the cloud result in models with a resource footprint larger than most edge devices can handle. The common practice is to run the trained inference model in the cloud as well to overcome the resource limitations. The edge device offloads the inference task to cloud and receives the result back from the cloud. While this enables deep learning inference on edge devices, there are some drawbacks to this method.

Latency
Some application require low latency to perform. Offloading the inference task to the cloud can dramatically increase the inference latency [49].

Bandwidth
Offloading the data to the cloud can saturate the available bandwidth of the edge device. Devices that have limited bandwidth may not be at liberty to send all the data to the cloud at all times.

Costs
To offload edge inference tasks to the cloud, (a cluster of) servers must be available for the edge devices. The cost of running these servers can increase when more and more edge devices uses the application that offloads the task.

Privacy
Some data needed for an inference task can be privacy sensitive, for example heart-rate sensors and health sensors. Running the inference models at the edge makes the need of sending all the data of the edge devices superfluous.

3 Related Work

In this chapter we discuss existing research done in similar fields. We start a review of the developments in Edge Inference. The structure of the related work follows a tree structure as can be seen in Figure 3.1 with a breath depth search. At the top the overarching term DNN Edge Inference is located. The related topics and subfields branch out towards the leaves. At the bottom of Figure 3.1 in the box *Memory Aware* solutions, the proposed solution of this thesis, MASA, is located. Although MASA is will be discussed in chapter 6 and not in this chapter, its position in relation to prior research is relevant.

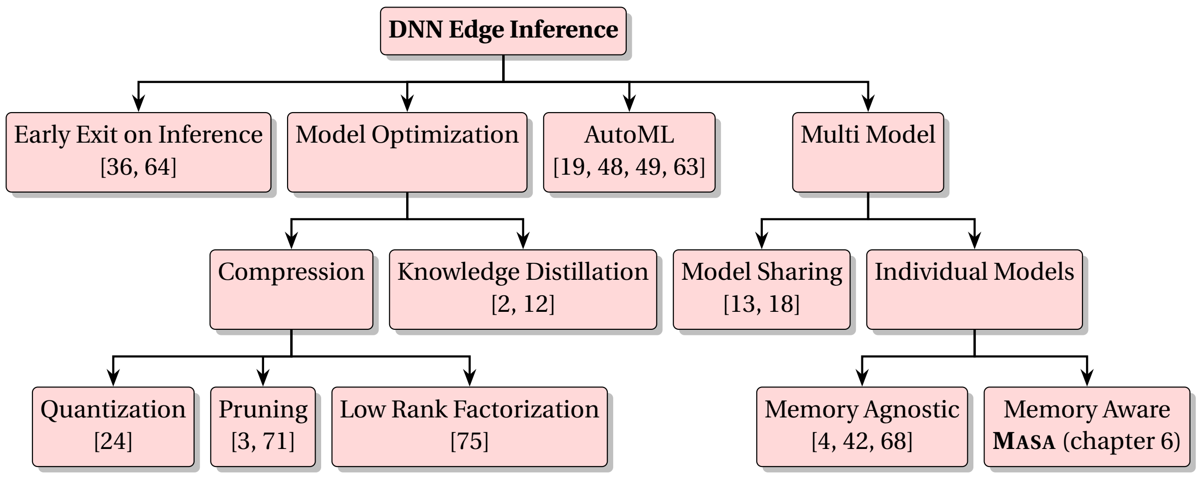


Figure 3.1: Related work hierarchy

3.1. Edge Inference

Large DNN models are trained in large cloud cluster reduce the training time. Edge devices are resource constraint by nature. Edge devices do not have the liberty to add more resources to improve the performance of inference models. There are various challenges when running inference model of large DNNs on edge devices.

3.1.1. Hardware

The types of edge devices are numerous with each different specifications in terms of RAM, CPU, the availability of co-processors and so on. Wu et al. [66] shows that the distribution for mobile devices alone if heavily segmented, with the most commonly used SoC (System on a Chip) at less than 4% market share. The heterogeneous nature of the mobile devices in terms of hardware makes it challenging to create a single model that runs on a large part of the devices. This problem will only increase when more and more IoT devices will be deployed. Even using identical hardware does not give performance guarantees. The performance of edge devices differs on the environments and scenarios in which the edge devices are used. The ambient temperature has impact on the performance. Wu et al. [66] showed that devices with identical hardware configurations show significant variability in terms of inference accuracy.

3.1.2. Models and optimizations

The latest rapid research of new DNN architectures and models compression techniques such as pruning, knowledge distillation, and quantization have led to an explosion of the number of deployable models. After including the numerous types and configurations of hardware, the search space become immense. The heterogeneous nature of the edge inference landscape makes it a challenge to efficiently pick the right model with the right optimizations for the given edge devices.

The size of the DNN models have grown significantly over time. The recent Natural Language Processing (NLP) models show (figure 3.2) an increase in the number of parameters. The move towards trillion parameter models [54] will be a serious challenge for Edge Inference. The growth of the model sizes is exponential (figures 3.2 and 3.3) while the growth of the processor speed is weakening. This means that the hardware cannot keep up with the growth of the DNN model size.

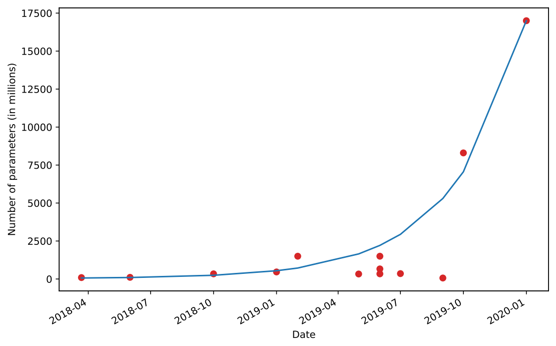


Figure 3.2: Growth of NLP models in number of parameters. This figure was adapted from a similar image published in [57] and based on [10, 15, 25, 30, 37, 38, 52, 53, 57, 58, 72]

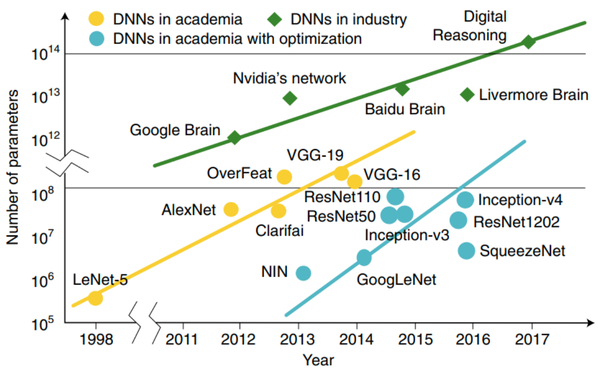


Figure 3.3: Growth of DNN in terms of number of parameters [70]

IoT in particular will have a challenge with hardware keeping up with the growth of the DNN models. Life-cycles of IoT-devices are already longer (> 10 years, [39]) than cloud systems. The increasing attention within the IoT field of research to extend device longevity will only widen the gap between the hardware-gap between IoT and cloud systems [43, 61].

3.1.3. DNN properties

DNNs exists with various architectures and various configurations. Often they are ordered in layers, for example convolutional layers of fully connected layers. The layers have each their own purpose and characteristics. Cheng et al. [7] notes that for deep CNNs most of the parameters are located in the fully connected layers, while most of the FLOPS are done in the first set of the convolutional layers. This means that CNNs require large amounts of computational power in the convolutional layers and requires large amounts of memory in the fully connected layers, as described by Lane et al. [31]. The compression of optimization for DNN is dependent on the types of layers that build op the DNN. The performance characteristics and resource requirements of a DNN is dependent on the composition of the different layers.

3.1.4. Trade-offs

Since the rise of deep learning, the number of parameters in the DNNs have increased [70]. The relationship between the number of parameters over time is exponential. With resources on edge devices being constraint, trade off exist when DNNs are compressed and optimized to run on target less performed than the cloud. Some application require low latency inference. Requiring low latency will lower the accuracy of the model. Canziani et al. [5] shows that there is a hyperbolic relationship between inference time and accuracy. A small increase in accuracy leads to a huge increase of computational time. [5] states that the number of parameters in a model is fair indicator of the estimated inference time.

Deep networks are used for more and more complex tasks. Using more layers does not always lead to a higher accuracy. Zhang et al. [74] shows that the performance of a network can become worse of to many layers are added. To allow DL inference models to run on edge devices, the models are compressed more and more to adhere to the constraint resources. This leads to desired specification of the trade-off between the accuracy and the compression rate [45].

3.2. Early Exit on Inference

A technique used to speed up inference time is Early Exit on Inference (also called DNN Right-Sizing). With this approach additional branches are added to a DNN. When the network is confident enough at a branching point, it can decide to exit the network early and skip the last remaining layers from execution. Teerapittayanon et al. [64] introduces Early Exit on Inference with Branchynet. Teerapittayanon et al. [64] show that for major networks (AlexNet, ResNet, LeNet) the inference speed can be increased when trained on the datasets MNIST and CIFAR-10. Early Exit on inference has influence on the inference speed/accuracy trade-off. The accuracy drops when inference speed if preferred over accuracy.

Li et al. [36] created a framework called Edgent. With Edgent the authors attempt to use Early Exit on Inference on low devices with limited resources. With the use of co-inference of an edge server, the set latency requirements were met. The decrease of latency caused an decrease of accuracy.

3.3. Model Optimization

Different techniques are used to optimize DNN models. Using these optimizations makes it possible to run DNN on edge devices. The resource footprint can be reduced using general model compression techniques, inference latency can be reduced by using early exit on inference, and network and hardware specific optimizations can be used to improve performance of the networks.

3.3.1. Compression

State of the art deep neural networks rely on millions and sometimes billion of parameters. A part of the parameters in large network have little to no contribution to the final output. These parameters can be seen as redundant and the network can be compressed accordingly. Model compression has the categories (1) knowledge distillation (2) quantization (3) pruning and (4) low rank factorization.

Knowledge Distillation

Knowledge distillation is the technique where a smaller network is trained to achieve a similar output as a high performance large network. Aguinaldo et al. [2] shows that GANs can be compressed significantly while achieving better FID scores than 'regular' knowledge distillation techniques. While compression rates of 1669:1, 58:1, and 87:1 can be achieved without losing accuracy, it should be noted that this is only compared to the regular student techniques, not to the performance of the teacher network. This level of performance is only achieved by the new student network when the compression ration drop to 13:1 and 4:1.

Eth et al. [12] combines quantization and knowledge distillation to compressed models. This *Quantized distillation* not only trains a student networks with a shallower network than the teacher, but is quantized as well. Eth et al. [12] shows that using Quantized distillation a compression ratio of 5:1 can be achieved while sacrificing minimal in terms of accuracy. The authors compare their results to a student network where quantization has applied after training the student network.

Quantization

Quantization method (like fixed point representations) are already often used the minimize the representation size of the data. Zero skipping, pruning and other data reduction method often result in additional complexity and irregularities in the hardware.

Jiang et al.[24] uses graph optimizations and quantization to increase the performance of CNNs in terms of inference speed. The authors are able to increase the inference speed and lower the memory, the sacrifice in accuracy is significant, up to 5%.

Pruning

A method of model compression is pruning. This field is widely studied. With pruning the number of parameters in a DNN is reduced by removing redundant layers, connections and removing parameters that have little to no contribution to the result. The main idea of pruning is to remove parameters that little to no influence to the result. Yang et al. [71] proposes a mixed pruning method that uses both filter-pruning and weight-pruning with a minimal drop in accuracy.

Both filter pruning and weight pruning is used (mixed pruning) by Yang et al. [71] to compress CNN models. First filter pruning is done where weak channels and filters are pruned. Secondly weight pruning is applied where weights below a certain threshold are pruned. The mixed are evaluated on the MNIST and CIFAR-10 dataset in comparison to the LeNet-5 and VGG-16 models. Compression rates of 13:1 and inference speed up of 3:1 are achieved. The results [71] show that weight pruning is the main contributor to the amount of achievable compression while the filter pruning is responsible for the speed up of inference time. The loss of accuracy is around 1.5%

Pruning can be done with different weights with respect to the layers of a network. The weights can be pruned uniformly, where all the layers apply the same threshold the force weights to zero, or it can be done in a non-uniform way. Ashouri et al. [3] shows that using a have the value of the threshold increase towards the end, or have the threshold relative to the range of the weights of the layers, sometimes have better accuracy. Ashouri et al. [3] shows that the way the threshold is chosen and applied depends on the type of the network.Ashouri et al. [3] only looked tested the methods at CNNs and uses a top-5 accuracy score.

Low-rank factorization

Computations in a convolutional layers are the majority of the computations in a CNN. These kernel operations can be represented a 4-dimensional tensor operation. Exploiting the possible redundancy in the tensor by using tensor decomposition's will speed up the convolutional layers. In some degree, this holds for the fully connected layers as well. Modeling the fully connected layers as 2-dimensional tensors suggests these layers will benefit in speed up by applying tensor decomposition's.

Zhang et al. [75] used a low-rank approximation of feature vector to reduce the amount of parameters in lower the inference time. Yu et al. uses a combination of pruning and QR decomposition's to reduce the number of parameters in both the convolutional and fully connected layers. This allows for high compression rates with minimal sacrifices to the accuracy.

Sparse networks

Most state-of-the-art DNN networks are using dense networks. This increases the number of parameters into multiple millions. A significant part of the parameters are either redundant or do not impact the result very much. Using sparse networks makes the DNNs more information compact. Sparse DNN can be achieved in one ore two ways: (1) using a sparse network from the start at the training phase or (2) sparsify the a trained network using for example pruning to create a sparse DNN for the inference phase. Both method have the benefit that the number of parameters are (greatly) reduced.

A way to train sparse networks is using a method called Sparse Evolutionary Training (SET) [44]. Using this technique the fully connected layers of DNN are replaced by sparse layers generated by SET. Mocanu et al. [44] claims that the number of parameters are reduced quadratically, without sacrificing any of the accuracy. A critical note is that fact that Mocanu et al. [44] evaluated only a single dataset (Cifar-10). The network used in this paper is very small. The number of parameters is around 9 million while that state of the art networks use around 557 million parameters Huang et al. [21]. This results in a difference in accuracy of 88% and 99% respectively. Their claim that their SET-CNN "outperforms" the conventional CNN with reduced number of parameters seems without a good foundation. It would be interesting if the same results can be achieved with large state-of-the art CNNS.

3.3.2. Network Specific

The inherent trade-off between accuracy and latency causes the creation of light-weight, efficient architectures [8, 20, 56]. Instead of applying optimizations on existing networks, the structure and working of networks can be altered. Xiangyu el al. [76] introduces with Shufflenet two new operations to reduce the computational cost: channel shuffle and group convolutions. The group convolutions reduces the complexity of the $1x1$ convolutions while the channel shuffle improves the information flow when the group convolutions are performed. A downside to ShuffleNet is that the shuffle operation add additional operations. Huang et al. [20] introduces a dense network with grouped convolutions. Additional Huang et al. tries to eliminate redundancy in the learned features. Although the results show that the number of FLOPS are reduced, while retaining comparable accuracy in comparison to other CNNs, the impact relies on the two hyper-parameters condensation factor and number of groups. This means that the quality of the network is dependent on the knowledge of the human tuning it.

3.4. AutoML

The conventional way of model optimization relies on handcrafted solutions. The heterogeneous hardware of edge devices and the increasing number of optimizations gives more possible solution for a problem. The large search space makes matching the right model with the right optimization given a hardware configuration a difficult task. Model selection is used to overcome this problem.

The mobile inference platform MODI, attempt to dynamically select inference models at run time [48]. Ogden et al. [48] shows the difference in run time on different devices and the impact of different compression techniques but does not expose the actual selection procedure. Ogden et al. [49] uses a Service Level Agreement (SLA) to dynamically select the model for an inference task. The model is selected based on the time budget given by the SLA. The model is selected solely on both the time budget and the estimated model accuracy. Taylor et al. [63] uses a series of KNN models to decide what pre-trained model to use based on the feature found in the model. Taylor et al. runs a so called pre-model to determine the best DNN for each image. Reinforcement learning is used by He et al. [19] to automatically select the best model compression technique for the given resource constraints. The agent is trained to predict the best action for each layer in the to be optimized network. As the results of He et al. show an improvement of compression ratios and inference speed, although is it only compared to handcrafted models by human experts. This shows the need for an automated way of matching models with optimizations. Model selection is a natural next step for Edge Inference. While model selection gets more attention, the research often is restrained to model selection of CNNs [19, 48, 49] instead of the general case of DNNs.

3.5. Multi Model

The previous studies in the above subsections has a focus on single model DNN inference. Running more than one model on a single edge device is a realistic option to leverage the distinct features the different models can provide. The detailed comparison is summarized in Table 3.1 with additional emphasis on the evaluated workload scenarios and hardware architectures.

3.5.1. Memory-aware DNN Inference

As the size and complexity of DNN applications have grown, their need for computing resources and memory has increased tremendously. It is particularly vital to manage and control the execution of DNNs on edge devices that have limited RAM. Existing solutions resort to model compression [44, 71], quantization. [24] , or network pruning [47, 71] to reduce the memory demands of DNNs. DeepMon [22] and DeepCache [69] are mobile deep learning inference frameworks which accelerate CNNs execution using cache mechanisms for processing convolutional layers. Jiang et al. [24] use graph optimizations and quantization to increase the performance of CNNs in terms of inference speed. Yang et al. [71] propose a mixed pruning method with a minimal accuracy penalty, which reduces the number of parameters in a DNN by removing redundant layers. PatDNN [47] is a pattern-based DNN pruning framework, including compressed weight storage, register load redundancy elimination, and parameter auto-tuning. The common theme of aforementioned approaches is to tradeoff the accuracy for the resource efficiency for a single DNN, whereas MASA manages the memory requirements of multiple DNNs without altering the network structure and its resource demands.

3.5.2. Multiple DNNs

Running multiple models on the same device at the same time or sharing resources with other processes is an effective but challenging way to multiplex the limited available resources. Several recent studies [4, 13, 33, 41, 42] propose solutions to efficiently run multi-DNN inference such the energy consumption, response time and accuracy can be optimized. NeuOS [4] minimizes a three-dimensional space, including latency, accuracy, and energy at the layer granularity for multi-model execution in autonomous systems. DeepEye [42] optimizes the execution of multiple CNN models by interleaving the execution of convolutional execution and fully-connected layers for memory-limited wearable devices. The study in [41] dynamically determines which DNN should be selected to execute a given input by considering the desired accuracy and inference time. NestDNN [13] allows for models to run concurrently and scale down to a model with smaller resource demands by using multi-capacity models and surrendering accuracy. MCDNN [18] relies on model sharing of variants with the same base model to efficiently run the same model type with different tasks. DART [68] is a pipeline-based real-time scheduling framework with data parallelism. DART groups a subset of DNN layers (task-level stage) and assigns them to workers of CPUs and GPUs to minimize the task response times by balancing tasks over resources while respecting time constraints.

Method	**Multi-DNN**	**Stochasticity**	**No DNN Similarity**	**Memory aware**	**Architecture**	
					GPU	CPU
MCDNN [18]	✔	✗	✗	✔	✔	✔
DeepEye [42]	✔	✗	✔	✗	✗	✔
NeuOS [4]	✔	✗	✔	✗	✔	✔
NestDNN [13]	✔	✗	✗	✔	✗	✔
DeepMon [22]	✗	✗	–	✔	✔	✗
PatDNN [47]	✗	✗	–	✔	✔	✔
DeepCache [69]	✗	✗	–	✔	✔	✔
DART [68]	✔	✗	✔	✗	✔	✔
MASA	✔	✔	✔	✔	✗	✔

Table 3.1: Overview of prior-art.

Summary

Edge Inference was the focus of numerous studies in the recent years. Several options exist to use trained models on the edge.

- Reducing the resource footprint of trained models by model compression shows promising results with the downside that the model accuracy is always affected.
- The matching of a set of trained models with a hardware configuration can be make simple by using techniques such as AutoML. While it automates the search through of the possible solution, it lacks adaptiveness to changing resource availabilities and support for multiple models.
- Deploying multiple models at the same edge device can be done by exploiting similarities in the models. This solution requires the deployed models to have similar features and thereby cannot be applied to all models.
- Deploying multiple models that have no similarities is still a field where the available solutions is limited. There exist some solutions like DeepEye but these are not aware of the available resources (memory) and thus perform not well under changing environments.

The aforementioned studies accelerate the inference time of multi-DNNs. From the spectrum of studies it can be seen in Table 3.1 that memory-aware multi-DNN is poorly represented.

4 Deterministic Scheduling

Finding a good schedule for (multiple) DNNs is not a trivial task. Networks are not equal. They need different resource requirements. Networks arrive at different random moments and more. How can the properties of DNNs be used to find a more efficient schedule? As scheduling is to find an order of execution of elements, can break down a DNN to give more flexibility by adding more options. Finding an optimal schedule when there are few options (inflexible) is a relatively easy task. This can be done by scheduling the DNNs as single blocks. This however does not account for the different resource needs of each layer, this is resource inefficient. Breaking down a DNN in smaller blocks makes it possible to take the individual requirements of the layers into account. This gives a scheduling algorithm more options in terms of order of execution and makes it harder to find the optimal schedule. It can be seen that scheduling on a smaller granularity gives more flexibility in the schedule but makes it harder to find the optimal schedule. To find how far a DNN needs to be decomposed, it is important to understand the structure of a DNN and how data is used.

4.1. Anatomy of a DNN

A DNN can be defined as a composition of layers. Often only the most important layers are used to describe a network. Common DNNs such as a CNN start with a set of convolutional layers followed by fully connected layers. For instance, the object detection network YOLO [55] consists of only convolutional layers and AgeNet [34] consists of convolutional layers followed by fully connected layers. The convolutional layer from the CNN is often followed by a Relu, pooling, and normalization layers. The fully connected layer is also often followed by a Relu (Rectified Linear Units) layer. Depending on the network, the choice and composition of the layers can vary.

The layer is one of the base components of a DNN. Looking at the a layer, it has input data, the layer itself, and output data. The computation of the layer itself relies on two items: the input data, that is provided by the previous layer, and the layer parameters, often loaded from disk. The output data follows from applying the computation in the input data. These semantics of DNNs gives some constraints regarding the type of actions that need to be taken to compute a layer and the order in which they should be executed.

The computation is dependent on the input data and the layer parameters. It follows that the parameters needs to be loaded before execution of the layer. It also follows that the parameters needs to stay in memory during the computation, hence the parameter may only be unloaded after the layer execution. This gives the following partial order *load* < *execute* < *unload*. This can be rewritten as $L_i < E_i < U_i$ where L_i, E_i, and U_i are the loading, execution and unloading task respectively for a given layer i. As the input data is provided by the output of the previous layer, it follows that the current layer i can only be executed if the previous layer $i-1$ is finished: $E_{i-1} < E_i$. The layer semantics if the layers in DNN gives the following partial orderings:

- $L_i < E_i < U_i$
- $E_i < E_{i+1}$

The semantics of a DNN layer gives us two partial orderings of the tasks within the same network. (1) The loading, execution, and unloading tasks of the same layer are ordered as $L_i < E_i < U_i$, (2) the execution tasks in the same networks are partially ordered as $E_0 < E_1 < \cdots < E_n$. Using these actions we can split a single networks of for example 10 layers into 30 tasks. When a task is dependent on the outcome of another task, it has a dependency relation (Equation 4.1) with that task.

$$\phi(A, B) = \begin{cases} 1 & \text{if task A uses the memory of task B} \\ 0 & \text{otherwise} \end{cases} \tag{4.1}$$

Given the dependency relation and the partial orderings, a dependency graph can be constructed. The dependency graph shows the required set of tasks B that needs to be executed before task A can start. From this dependency graph, a set of topological sortings can be calculated using Kahn's algorithm [26]. By decomposing the single DNN in smaller tasks, the increased number of topological sortings gives more flexibility in the execution of the DNN.

4.1.1. Modeling a DNN

Choosing the right granularity for the task size is important. The larger the task (use the network as a whole), the easier the scheduling becomes. The smaller the granularity of a task (sub-layer), the more difficult the scheduling becomes. When the granularity is too large, potential resources are wasted. This can be seen in Figure 4.1, the resource required of tasks uses the larges required resource in the lifetime of the task. When the task is as large as the network itself, the required resources for that given task is equal to the largest layer in the network.

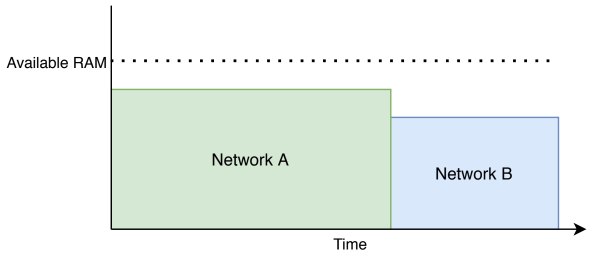


Figure 4.1: Using a task granularity that is too large causes network A and network B to run sequentially.

Using a smaller task granularity allows for a more fine-grained approach in terms of scheduling. Figure 4.1 shows that the area of the resource requirements of the individual layers is less than the area when network A or B is depicted as a single task. Figure 4.2b shows that the smaller task size allows for networks to be executed at the same time.

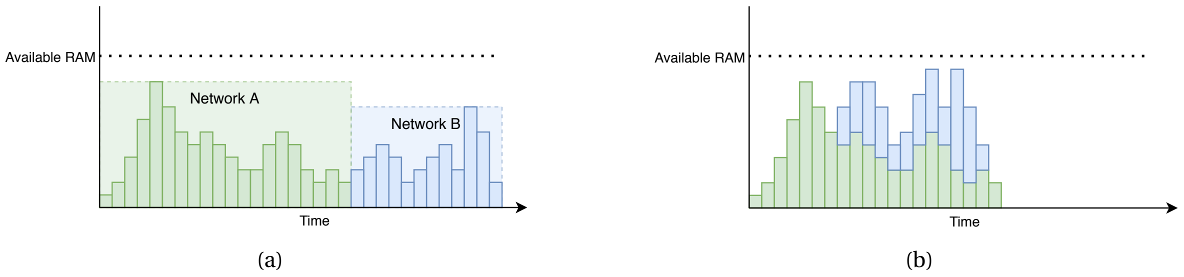


Figure 4.2: A smaller task granularity allows for concurrent network execution, within the bound of the available RAM.

4.1.2. Directed Acyclic Graphs

By defining structure of DNNs in terms of dependency graphs, we can derive some properties based on that. Dependency graphs are a type of Directed Acyclic Graphs (DAG). This means the properties of DAGs apply to the dependency graph of the DNNs. Using the semantics of Neural Networks we extracted the following properties:

- There exists a partial order between the loading, execution, and unloading of layer i: $L_i < E_i < U_i$
- There exists a partial order between the execution of layer i and layer $i+1$: $E_i < E_{i+1}$

Using this we can construct a generic dependency graph for a random 4-layer networks. We start by adding an initialization I task to construct the structure and placeholders of the network. All loading tasks are dependent on the initialization task, after all data has to be loaded in some data structure. The partial order of the tasks of the same layer defines the dependencies between $L_i < E_i < U_i$ for layer i. The partial order of the execution tasks between the layers defines the dependencies $E_i < E_{i+1}$. Using these rules we construct the partial dependency graph.

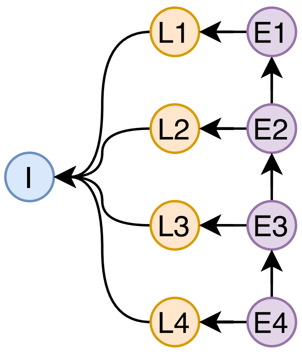
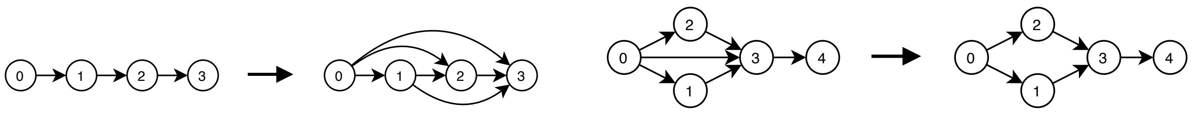


Figure 4.3: Example of a partial dependency graph of a four-layer DNN

Dependency graphs can be reduced using the transitive reduction algorithm [17]. Transitive reduction is the inverse operation of transitive closure [51], where for each vertex a direct edge is constructed to any other reachable vertex. With transitive reduction, the smallest graph $G' = (V, E')$ is constructed with a directed path from i to j in G' iff $(i, j) \in E$. For acyclic graphs both the transitive closure and the transitive reduction results in a unique graph [17].



(a) Transitive closure (b) Transitive reduction

Figure 4.4: Examples of transitive closure and transitive reduction

The transitive operations (closure and reduction) do not change the reachability of the graphs. The graph produced by transitive reduction is a subset of the input and for the transitive closure the inverse is true. The set of topological orders is also not affected by the transitive operations, since the edges remove by transitive reduction where not a viable option to consider in a topological order.

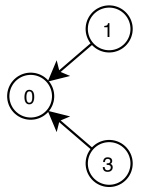
All dependency graphs have topological orderings. A topological ordering of a dependency graph (or DAG) $G = (V, E)$ is an ordering of the nodes $v_0, v_1, \ldots, v_n$ such that every edge(v_i, v_j) holds $i < j$. Topological orderings in dependency graphs are often not unique. This means that each dependency graph there is a set $\mathcal{T}$ of topological orderings. The cardinality of the set $|\mathcal{T}|$ is a measure of the number of topological orderings for a given graph G.

Add dependencies

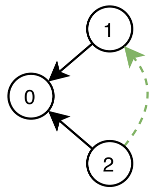
Another operation is adding dependencies in the form of a new edge. By adding new edges to existing vertices, the dependency graph becomes less flexible. The cardinality of the set of topological orderings becomes smaller. Figure 4.5 shows that adding an edge between existing vertices can decrease the cardinality of the set of topological orderings.

Graph derivations

From the partial dependency graph we can derive other dependency schemes: *linear*, *bulk*, and *deepeye*. The bulk schema represents the default method of loading and executing a network, the linear schema represents the execution of a network with minimal memory usage, and the deepeye schema represents a multi-model execution method [42]. For a four-layer network, by adding three additional edges a linear dependency graph can be derived from the partial graph.



(a) An example three-vertex DAG where the cardinality $|\mathcal{T}|$ of the set of topological orderings = 2, namely $\{(0,1,2),(0,2,1)\}$.



(b) Adding an edge from vertex 2 to vertex1 reduces the cardinality $|\mathcal{T}|$ of the set of topological orderings to 1, namely $\{(0,1,2)\}$.

Figure 4.5: Effect of adding dependencies to existing vertices.

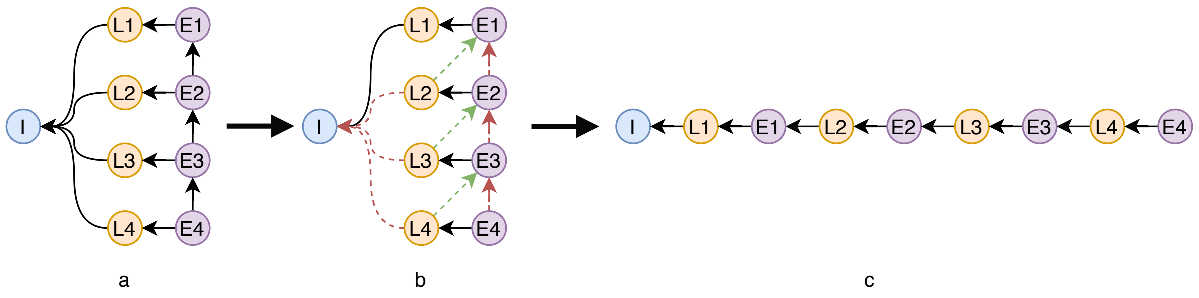


Figure 4.6: By introducing additional edges in the partial graph and after the application of the transitive reduction algorithm, the linear dependency graph is the result.

In Figure 4.6 the partial graph is seen in sub-figure a. After adding additional edges (green arrows) a transitive reduction is done (red arrows). After re-aligning the nodes, the known linear dependency graph is the result.

To construct the bulk dependency graph we start with the partial dependency graph. Following the constraints of bulk loading and execution (load all layers before execution) additional dependency are added (Figure 4.7b). Redundant dependencies can be removed using the transitive reduction algorithm. The result is the bulk dependency graph Figure 4.7c.

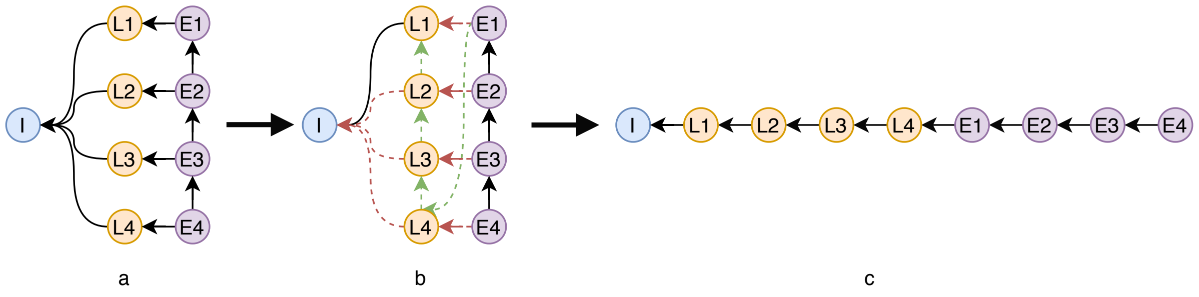


Figure 4.7: By introducing additional edges in the partial graph and after the application of the transitive reduction algorithm, the bulk dependency graph is the result.

Finally DeepEye dependency scheme can be derive from the partial graph. As can be seen in figure Figure 4.8, the same actions are used. First we introduce additional constraints that follow from the DeepEye algorithm. This means that the convolutional layers are loaded and executed while the fully connected layers may be loaded simultaneously. After applying the transitive reduction algorithm, the result is the DeepEye dependency graph.

From Figures 4.6 to 4.8 can be seen that the all are derivations of the partial dependency graph by adding additional constraints and applying transitive reduction. Since adding dependencies reduces $|\mathcal{T}|$ and transitive reduction does not increase $|\mathcal{T}|$ we can say that all the derivation graphs (linear, bulk, and deepeye) have less options in terms of order of execution, hence $|\mathcal{T}_{partial}| > |\mathcal{T}_d|$ where $d \in \{linear, bulk, deepeye\}$.

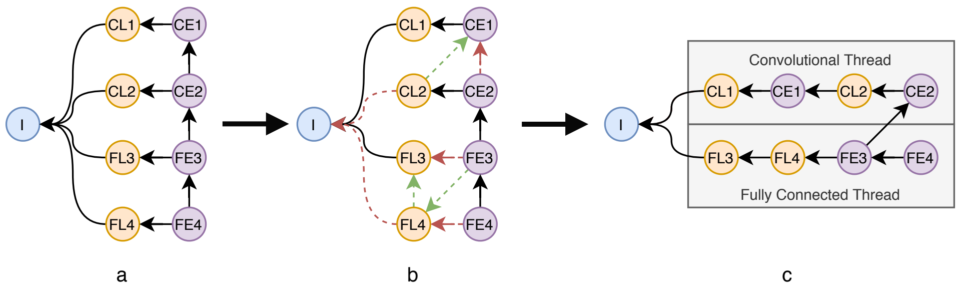
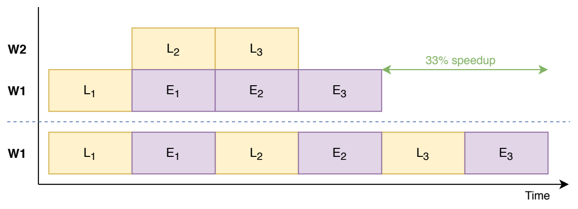


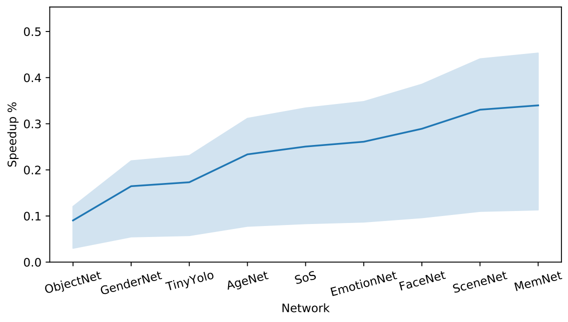
Figure 4.8: By introducing additional edges in the partial graph and after the application of the transitive reduction algorithm, the DeepEye dependency graph is the result.

Utilizing multiple threads

An execution task can only start after the related loading task has finished. While this dependency restricts task E_i from running while task L_i has not finished, starting L_{i+1} is not restricted by the dependency graph. The same holds when task E_i is running, we can also start task L_{i+1} simultaneously when following the dependency graph. In Figure 4.9a can be seen that by adding a additional worker (thread) dependency-free tasks can start next to other tasks. This allows for layers to be pre-loaded, loaded before they are actually needed. A downside of pre-loading or running multiple tasks concurrently is the increase of memory. When the available memory is sufficient, pre-loading can be effective in speeding up the inference processing time.



(a) Example of a three-layer network where adding a second worker (top schedule) creates a speedup of 33% in comparison to a schedule with one worker (bottom schedule).



(b) The potential speedup differs per network. The ratio of duration of loading tasks vs execution tasks is an important factor that impacts the potential speedup. The speedup is defined by the relative difference between the duration of the optimal schedule and the duration of the naive bulk schedule.

Figure 4.9: Utilizing multiple workers can speed up the inference process. Note: the examples in the sub-figures above are not limited by memory, hence the available memory is infinite for these examples.

The amount of the speedup that can be gained by adding additional workers (threads) depends on the networks and the ratio of loading duration vs execution duration. Figure 4.9b shows that the speedup differs per network. The speedup is defined by the relative difference between the duration of the optimal schedule and the duration of the naive bulk schedule.

4.2. CP-scheduling

Task scheduling can be modeled as a combinatorial optimization problem. Given the set of tasks, find a ordering such that the objective function (for example makespan) is minimized. Constraint programming (CP) is the paradigm for solving combinatorial problems using constraints. This is called a Constraint Satisfaction Problem (CSP). A CSP is defined as a set of variables $X_0, X_1, \ldots, X_n$ and a set of constraints $C_0, C_1, \ldots, C_m$. The domains $D = D_0 \times D_1 \times \ldots \times D_n$ are defined for each variable X_i.

To solve a CSP, all the constraints for each variable has to be satisfied. This gives the search space for solutions of a CSP to be all the feasible solutions. Finding the optimal solution results in solving the given objective function $f : D \to \mathbb{R}$.

In general a combinatorial problem can be modeled using constraints and solved by a general purpose solver like IBM ILOG CP optimizer [29] or the OR-Tools CP-Sat solver[1].

4.2.1. Complexity

Describing our scheduling problem in the form of a Constraint Satisfaction (SAT) Problem gives information about the complexity of the problem. Using the Cook-Levin theorem [9], is can be shown that the CSP is NP-complete. Our scheduling problem can also formulated using the Hamiltonian Path problem and the BinPacking Problem. For the shortest Hamiltonian Path problem, we need to generate a directed graph from the set of topological ordering. Every tuple in a topological ordering describes an edge between two vertices. As the directed graph is the combination of all topological ordering, we know that a Hamiltonian path must exist. The weights of the edges is the time to execute a task. The Hamiltonian Path problem can be reduced to the SAT problem: SAT $\leq_p$ Hamiltonian Path.

4.2.2. Formal definition

The constraints for the model can be constructed by examining the structure of DNNs. We have N tasks that need to be scheduled across W workers. Let $T = \{T_1, T_2, \ldots, T_n\}$ be all the tasks. Let $W = \{W_1, W_2, \ldots, W_w\}$ be all the workers. Each task has a start time S and a processing time P. From this we can derive the completion time C for each task.

$$C_i = S_i + P_i,\ \forall T_i \in T$$

From this object objective function can be minimized. We want the makespan of the tasks to be minimal, as is stated in Equation 4.2.

$$\min(C_{\max}) \tag{4.2}$$

As tasks are not fixed to any worker, we need to keep track of task-worker-assignment relation with the variable X.

$$X_{i,j} = \begin{cases} 1 & \text{if task } i \text{ is scheduled on worker } j \\ 0 & \text{otherwise} \end{cases}$$

Tasks can only be assigned once; we don't want to load or execute a layer more than once. This gives the first constraint: task must be assigned exactly once.

$$\sum_{w \in W} X_{i,w} = 1,\ i \in T \tag{4.3}$$

We know that the time of completion C_i is defined as the sum of the start time and the task duration. This gives the second constraint Equation 4.4.

$$C_i = S_i + P_i,\ i \in T \tag{4.4}$$

All tasks must be non-negative, since negative duration does not make sense.

$$P_i \geq 0, \forall i \in T \tag{4.5}$$

Tasks can only be assigned once to a worker. By summing the assignments of each task across all workers, we can be sure the the number of assignments is equal to one.

$$\sum_{i \in T} X_{i,w} = 1,\ w \in W \tag{4.6}$$

An assigned task is a time interval defined by the tuple $[S_i, C_i]$ with S_i and C_i as start time and completion time respectively. All intervals on the same worker must be non-overlapping. This is valid when the intersection of the intervals of the tasks are the empty set. This only has to be true for the intervals that are scheduled on the same worker w.

$$[S_i, C_i] \cap [S_k, C_k] = \emptyset,\ \forall i, k \in T : i \neq k \text{ and } X_{i,w} = X_{k,w} = 1, w \in W \tag{4.7}$$

[1]https://developers.google.com/optimization

We know that some tasks are dependent on other tasks. Naturally this leads the to constraint that task A must start before task B is task B is dependent on task A. Using the dependency relation function $\phi(a,b)$ of Equation 4.1.

$$C_i \leq S_k,\ \forall i,k \in T : \phi(k,i) = 1 \tag{4.8}$$

4.2.3. Simple memory model

From section Figure 4.1.2 we know that tasks can load parameters, execute the layer, and unload the layers. Both loading and executing a layer results in an increase of memory usage. Loading parameters from disk naturally increase memory usage as the parameters moved from disk to RAM. During execution of a layer, the intermediate results are kept in memory and this also increases the memory usage. We can simplify this by saying that each tasks has a direct memory usage of m_i for task i with $m_i \in \mathbb{Z}^+$.

Every task uses memory to execute. We need to make sure that the memory usage at any point in time does not exceed the available memory pool M_{system} of the system. To this end a constraint is needed to prevent over usage of memory. For each point in time τ_l, the summed memory usage m_i of each task i that is currently running must be lower than the system limit M_{system}. The direct memory usage of tasks for a certain point in time τ is saved in parameter M_{tasks} as can be seen in Equation 4.11.

$$M_{tasks} = \sum_{i \in T} (S_i \leq \tau_l \wedge \tau_l \leq C_i) * m_i,\ \forall l \in \mathbb{N} \tag{4.9}$$

4.2.4. Locked memory model

Modeling the current memory usage with only direct memory is a simplification of the real usage. Tasks of the same layer are highly related to each other, as can be seen in Figure 4.10. The partial order of loading, execution and unloading of the same layer exists because the execution tasks uses memory that is loading into RAM by the loading task. This results in the relation $\theta(A,B)$ where task A uses memory allocated by task B.

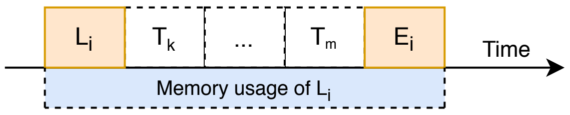


Figure 4.10: Example how task the memory allocated by L_i must be retained because task E_i expects it to be available.

Some tasks uses not only their own memory be also the memory of other tasks. This means that we can define a relation $\theta(A,B)$ between task A and B is task A requires the memory of task B

$$\theta(A,B) = \begin{cases} 1 & \text{if task } A \text{ requires memory of task } B \\ 0 & \text{otherwise} \end{cases} \tag{4.10}$$

To incorporate the locked memory into the CP-model we introduce a new interval parameter. Let a locked memory interval defined by $[D_{d,e}, E_{d,e}]$, where $D_{d,e}$ and $E_{d,e}$ are the start and end time of the interval where the memory is locked from task$_d$ to task$_d$. The memory usage of such an interval is denoted by $I_{d,e}$, where I is the amount of memory that is locked from task$_d$ to task$_d$. All locked memory for every point in time τ is saved in parameter M_{locked} as can be seen in Equation 4.11.

$$M_{locked} = \sum_{d,e \in T : \theta(e,d)=1} (D_{d,e} \leq \tau_l \wedge \tau_l \leq E_{d,e}) * I_{d,e},\ \forall l \in \mathbb{N} \tag{4.11}$$

We can use Equation 4.9 and Equation 4.11 to construct the constraint the keep the memory usage of all tasks below the system limit.

$$M_{system} \geq M_{tasks} + M_{locked} \tag{4.12}$$

Let $W = \{W_1, \ldots, W_n\}$ be the set of workers. Let $T = \{T_1, \ldots, T_n\}$ be the set of available tasks. Lets $T_{i,m}$ be the set of tasks that Each worker can process one task at a time. This result in the case that $S_{i,j} + P_{i,j} = C_{i,j}$

The CP-model is listed in Appendix A.

4.2.5. Scalability analysis of the CP-model

Using Constraint Programming to find the optimal solution can only be used on the edge devices in the online phase when it scales normally. To test the scalability of this solution a couple factors are tested. The parameters that can change during execution are: the number of layers in a network, the average task duration in the network, the number of workers in the schedule, and the available memory in the schedule.

First, what happens when the size of the network changes. A synthetic DNN is generated. For small networks (< 17 layers), the computational effort to find the optimal solution using CP is small. When networks becomes larger the needed computational time required to find the solution increases dramatically as can be seen in Figure 4.11.

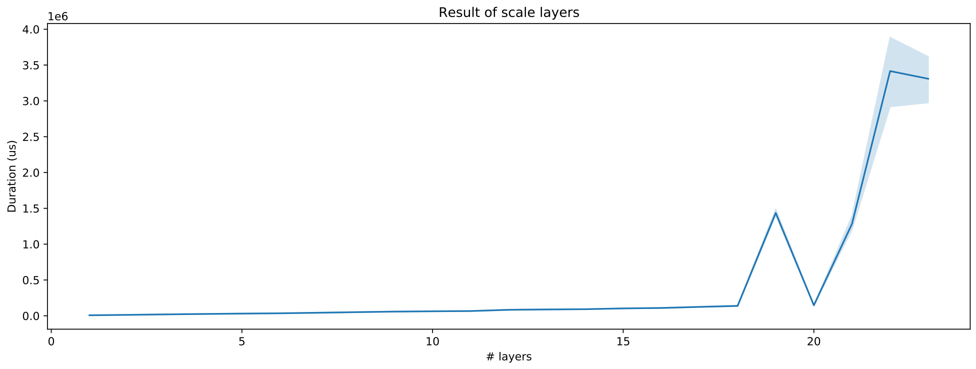


Figure 4.11: Increasing the number of layers in a network has a negative effect on computational time to find the optimal CP-solution. The number of repetitions for this experiment is 20.

As tasks can have different sizes of durations, it is interesting to the seen effect of the average task duration on the computational effort. To this end, a fixed size synthetic DNN was generated where the task size was increased after every run. Figure 4.12 shows that increasing the average task duration increases the computational effort linearly. Since multiple workers are used in the solution to speed up the execution of DNN, it

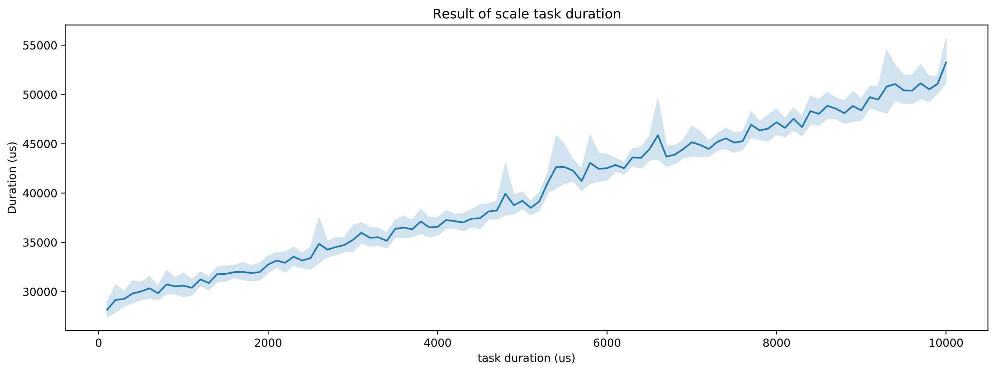


Figure 4.12: Increasing the average task duration in a network shows a linear relation with the required computational effort to find the solution. The number of repetitions for this experiment is 20.

is sensible to measure the effect of the number of workers used in the solution. A fixed size synthetic DNN is used and each run one worker is added to the schedule. This should, in theory, make the solution space larger since more options are possible. Figure 4.13 indeed shows that the computational effort increases when the number of workers increases with an exponential factor.

The available memory for the schedule is also a parameter that can be different for any given time. More available memory allows for more parallel execution of tasks. Figure 4.14 shows that the computational effort increases when the available memory decreases. This is also the case the number of workers is changed. The point of increase differs depending on the number of workers.

From the aforementioned figures can be seen that three out of four parameter scale exponentially. With these observations we must conclude that solving the scheduling problem in an online 'real-time' manner on

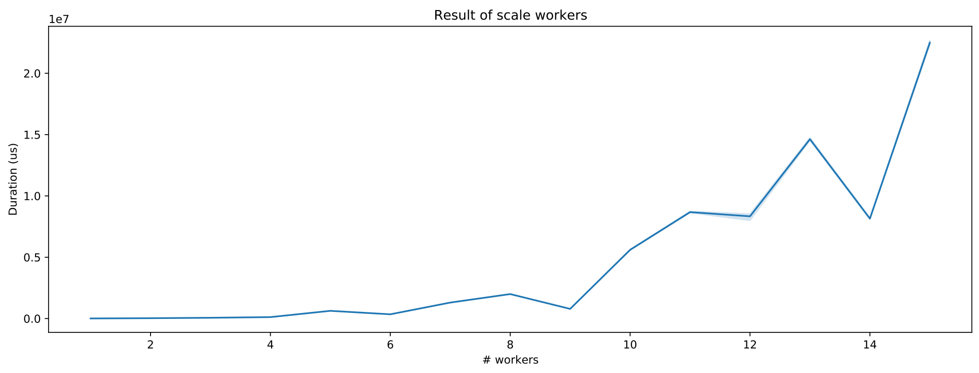


Figure 4.13: Increasing the number of workers in the schedule has an exponential effect on the required computational effort to find the solution. The number of repetitions for this experiment is 20.

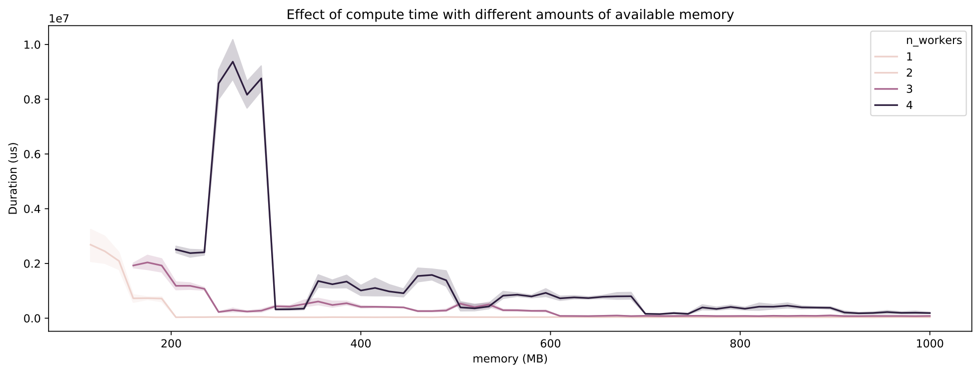


Figure 4.14: The memory available for the schedule has a negative impact on the computational effort. The number of repetitions for this experiment is 20.

an edge device is not feasible.

4.2.6. Robustness of the CP-model

Since the CP-solution cannot be run on the edge device, maybe we can calculate schedule for networks in certain situations beforehand. This comes down to the accuracy of the estimated parameters. Since the parameters, task duration and memory usage are benchmarked, these values have some uncertainty. The solution found using Constraint Programming is optimal for the given situation but is assume perfect information. The question arises about the robustness of the found schedule. It boils down to the following: how much can the benchmarked parameters deviate before the schedule gets a different ordering. If the relative order of the schedule does not change, we can still pre-compute the schedule and use on a later moment on the edge devices. If the relative order of the tasks in the schedule changes, the whole schedule needs to be re-computed. To measure the ordinal association of two schedules the Kendall rank correlation coefficient is used, the Tau-b in specific.

First the base schedule is computed using CP. This schedule is based on perfect information. Then the benchmark values are changed by 1 to 30%. This is done for both the positive and negative effect (this means the ranges $[0.7, 0.99]$ and $[1.01, 1.3]$). For each schedule alternative, the Tau-b value is calculated with respect to the base schedule. A value of 1 indicated that both schedules show a strong agreement. A value of -1 show that both schedules have a strong disagreement. In short, a value of 1 represents perfect similarity between two schedules and anything less than 1 shows the schedules are ordered differently. Figure 4.15 shows that the alternative schedule starts to deviate significantly from the base schedule when the benchmark is changed by as few as 1%. This shows that precomputing the schedules ahead of time is only possible when we act on perfect information. The solution found by Constraint Programming is not robust enough to be used to schedule ahead of time.

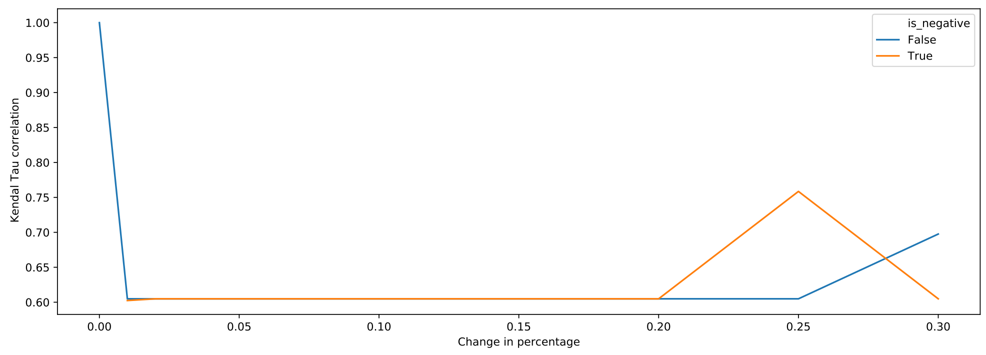


Figure 4.15: The effect on the schedule similarity when duration changes by a given percentage. Higher is better.

The aforementioned examples were about finding a schedule for a single model. Finding a schedule for a multi-model problem requires little augmentation to the setup. In the set of tasks two networks are created instead of one. This results in two disjoint dependency graphs. Since these could be executed fully in parallel, the set of possible tasks to executed at any given time is larger. When looking at a multi-DNN problem the solution scales shows the same trend in terms of scalability. When increasing the number of networks in the problem, the computational effort is significantly worse for problems with a higher number of networks where the total number of layers remain the same. In Figure 4.16 can be seen that a three-network problem with a total of 8 layers needs a lot more computational power to solve than for a two-network problem with the same total amount of layers. The same hold for the comparison of a two-network problem and a one-network problem. Since a multi-network problem has inherent parallelizable tasks, the number of possible schedules increases with each added network.

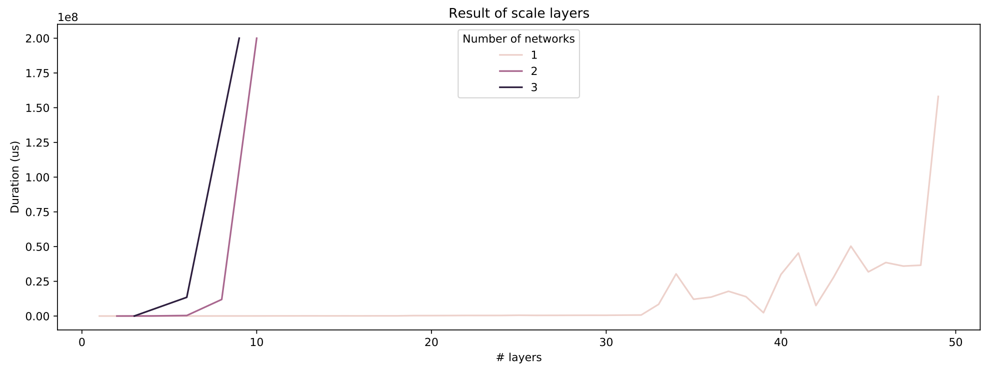


Figure 4.16: Effect on required computational effort when increasing the number of networks in the problem. The number of layers are the total number of layers in the problem; the sum of all networks.

Summary

DNNs can be decomposed by sub-layer tasks. This smaller granularity gives more flexibility in terms of scheduling. Modeling the DNN scheduling problem with Constraint Programming is not a feasible method for online scheduling on edge devices. The CP-model does not scale well with the numbers of workers and the size of the networks. The CP-model is also not robust to use for scheduling ahead as described in subsection 4.2.6. As it is an approximation of the real situation, the uncertainty of for example disk speed latency invalidates the found schedule. Scalability of the CP-model is a real problem when scheduling large networks.

- Deep Neural Networks can be decomposed in sub-layer building blocks.
- Layers can be described in three tasks: *load, execute, unload*
- Different layers have different resource needs. Convolutional layers have few parameters and require more execution time while fully connected layers have a lot of parameters and require less execution time but more memory space.
- Finding the optimal schedule using a CP-model is not a feasible solution.
- For online scheduling of DNNs on edge devices we need a different method to schedule tasks.

5 EdgeCaffe

DDNs are normally executed in a single run: first the whole model is loaded and a forward pass is done the calculate the inference result. This method is used in most major DNN execution framework like Caffe [23], MXNet [6], PyTorch [50]. Even when the target device is on in available resources, the whole model is used. Techniques like compression, pruning, and quantization are used to fit a trained model on an edge device. Using the aforementioned techniques diminishes the accuracy of the trained model. In this chapter we explore a DNN framework, EDGECAFFE, that offers partial execution of Deep Neural networks. First, in the architecture of EDGECAFFE is discussed in section 5.1. Secondly we explore in section 5.2 how multi-threading is applied in the framework. section 5.3 describes how the performance of the models in terms of resource usages can be profiled. Lastly section 5.4 explains the tools that are available within the framework and how it helps the developer. The EDGECAFFE implementation is open-sourced on Github[1].

5.1. Architecture

The EDGECAFFE frameworks consists of three parts: the core library, the testbed and a third party DNN Framework (in this case Caffe [23]). The third party DNN framework facilitates the actual computations needed for DNN inference. Theses functionalities are used by the core library to coordinate partial execution of DNNs.

The core library is in charge of the partial execution and the coordination between multiple networks. It keeps track of the progress of an inference request, the internal memory usage, and more.

Finally the testbed is used to profile inference runs, generate workloads and setup resource limited environments.

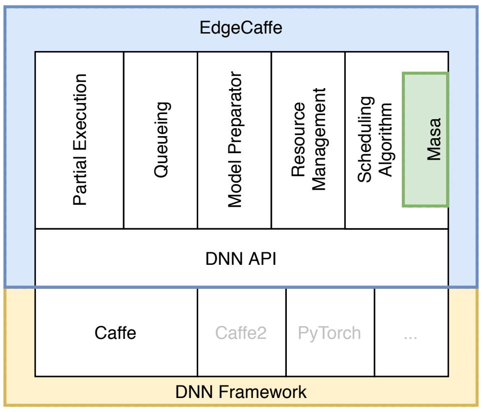


Figure 5.1: Components of EDGECAFFE

[1] `https://github.com/bacox/edgecaffe`

5.1.1. Core library

As the core library is in charge of coordinating the partial execution, numerous components are needed to facilitate this. The most important part of the core library is the execution engine. This makes it possible to execute a DNN partially. The actual (partial) execution of the DNN inference is done the DNN framework (in this case Caffe). Calls for this execution are done through the DNN API. This API makes it possible to swap the Caffe framework for a different library like PyTorch. The core library needs to be able to respond to new inference requests and dispatch the inference results to third party applications who request an inference to be done. To this end, the request handle exists. This component keeps track of all the inference requests arrive, are queued and are finished. The Network Storage stores the information for different DNNs that are used during execution. It stores information about the network structure and the location of layer files on the disk. The Task builder lets the system deconstruct a trained DNN into a set of tasks and their given dependencies. During execution it is important that the limited available resources are respected. Therefor, the resource monitor exists. This element keeps track of the allocated layers across all the networks and makes sure that is stays within bounds.

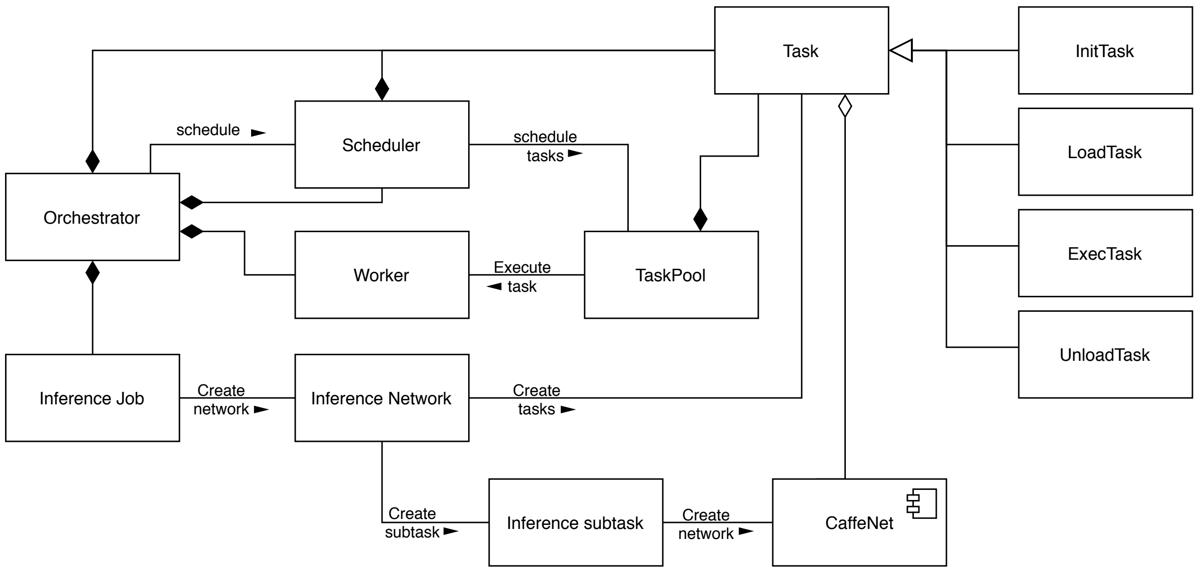


Figure 5.2: Architecture of EDGECAFFE

Execution Engine

This is the main component of EDGECAFFE. Tasks generated from trained DNNs need to be scheduled. The execution engine is in charge of this effort. 5.2 shows the components within the execution engine. A set of N workers are used to execute individual tasks. The scheduler creates an execution order based on the available tasks, their task-dependencies, the available resources, and the available workers.

Network Storage

When an inference request is made by a third party application, only the input data and the DNN tag name is given. This keeps the data linked to the request small and easy to copy. With the network tag, EDGECAFFE can retrieve all the required information of the network from the Network Storage. For example the network Storage keeps track of the actual location on disk of the parameter file for a given network. Additional information about the layers and network can be retrieved from the Storage like expected execution time for a certain layer or the required available memory.

Model Preparator

The model preparator is not used during the processing of an inference request. This tool can be used to prepare a new trained network to be added to EDGECAFFE. Freshly trained networks needs to be altered slightly for optimal performance in EDGECAFFE. The model needs to be split up in separate files, one for every layer. Besides that, a more detailed description of the network needs to be created. The model preparator can be used for all these tasks.

Task Builder
When a trained network needs to be executed on EDGECAFFE, it first needs to be broken down into tasks. The Task Builder uses the network description found in the Network Storage to create the needed tasks. The task builder also builds the dependency graph that describes the relationships between the tasks of a network.

5.2. Multi-threading

EDGECAFFE uses multiple threads to achieve concurrent execution of DNNs. When N workers are requested, $N+1$ threads are used for EdgeCaffe. The additional treads is used to keep track of the network state, handling incoming inference request, and more. This 'control' thread coordinate all the work done in EDGECAFFE. As can be seen in Figure 5.4, the 'control' thread includes a lot of separate components in the EDGECAFFE.

5.2.1. Worker thread

The workers are responsible of the execution of the scheduled tasks. A worker waits for tasks to be ready for execution and is idle otherwise, Figure 5.4. When a worker receives a task from the scheduler, the task is executed. After execution, the task is marked as finished and transferred to the finished task pool. Tasks have a reference to the Neural Network object. This data is shared across all tasks of the same network. Normally, running multiple entities at the same time that share data is prone to race conditions and memory corruption. Due to the dependency graph of a network, no tasks can use the same memory at the same time, the partial order preserved in the dependency graph does not allow for it.

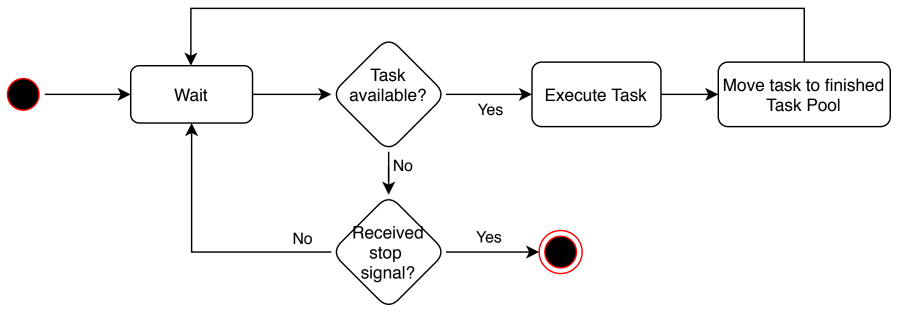


Figure 5.3: Flow diagram of a worker thread

5.2.2. Control-thread

To coordinate all the components the control thread exists. All the components, except the workers, live in the control-thread. This means that all components in the control-thread are run sequentially (as can be seen in Figure 5.4). EDGECAFFE first checks if new inference requests have arrived. If that is the case, the new inference requests are transformed in a set of tasks. The freshly created tasks are then inserted into the waiting queue at the same time. The tasks are inserted simultaneously in order to make algorithms like Smallest Network First. If the tasks were to inserted one by one, automatically a First Come First Serve regime is created. The next step for the control-thread is to check is any of the dependencies of the waiting tasks are resolved and therefor move those tasks to the ready queue. Finally, the thread checks for tasks that are finished. When all the tasks of a network are finished, the result of the network can be retrieved, the network can be deallocated, and the result can be dispatched to the application that initially requested the inference task.

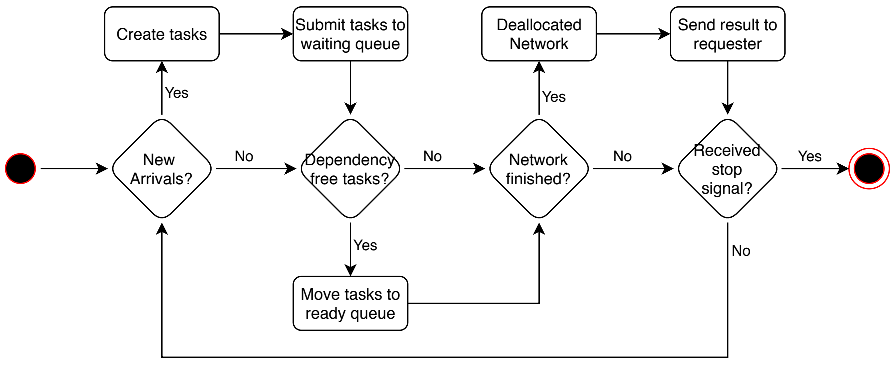


Figure 5.4: Components in EDGECAFFE that are controlled by the 'control' thread.

5.3. Profiling

In order to make memory-aware scheduling decisions, MASA needs accurate estimates of the memory requirements per layer. An under estimation of the memory usage causes the scheduler to over commit its memory budget and thereby likely causing paging and slowdowns. An over estimation of the memory usage can lead to under utilization of precious resources.

The memory profiler aims to precisely estimate the peak memory usage per layer which demands a finer granularity than existing well-known off-the-shelf profilers, e.g., Valgrind [46] and GPerfTools [16]. We hence resort to query the kernel via the `/proc/self/statm` interface to track the RAM usage of our process. Specifically, we execute each layer of a network in isolation by a single worker during profiling. To track the memory usage from the beginning of a layer task to its end, we spawn a parallel thread which continuously probes the consumed memory. As this piece of work is intrusive and has a negative impact on the performance of the system, it is not enabled in the framework when running normally. The profiler can be used to profile newly trained network and profile their runtime characteristics on a target device. The mean time between probing of the profiler thread is 7.12 μ s $\pm$5.5 μ s, which is significantly lower than the execution time of the smallest layer considered in our evaluation. Table 6.1 lists the storage space of each network and peak run-time memory usage. One can see that the active memory usage is multiple times higher than the storage space.

5.4. Tools

Within the framework there exist several tools that help the developer to make it easier to create and deploy partially executable DNNs. Some tools like the Model Splitter and the Network Descriptor are central for the working of EDGECAFFE. Without these two tools it would be very time consuming to create/prepare new networks. Tools like the Network Generator and Network Analyzer are targeted towards solving problems in networks and help testing the library.

5.4.1. Model splitter

The model splitter adapts a network trained with a normal framework (in this case Caffe) to be transformed conform the EDGECAFFE structure. The trained model is read into memory and the network structure is used how to determine how the parameter file needs to be split. A new parameter file is created for each layer of the network. This way layers can be loaded individually.

5.4.2. Network Descriptor

A splitted model is not the only required augmentation for a trained model to work on EDGECAFFE. The framework also needs a network description. This holds the additional information about the trained model. The network description contains the location of the parameter files as well as the additional information about each layer, such as the memory requirement and the estimated execution time of each layer. The Network Descriptor tool allows for this description file to be generated according the known information of the trained network.

5.4.3. Network Generator

Networks come in different structure, size, and types. As training a network is a costly process, we use pre-trained networks for this research. The number pre-trained networks available is limited. To overcome this limitation, the network generator was created. This tool can generate simple networks of an size and length. Networks can be generated where the first layers are fully connected layers followed by convolutional layers, how unusual they maybe. The ability to generate any type of network lets us more freely examine the properties of certain types of networks and layer combinations.

5.4.4. Network Analyzer

Sometimes it is useful to inspect a 'prepared' network. Small differences in the dependency graph can have a relative large impact. The Network Analyzer makes it possible to visualize the network dependency graph. Just as in the Core library (subsection 5.1.1), networks and splitting algorithms are available for use. The Model analyzer constructs the dependency graph from the tasks created by the core library. In this way we can easily see how the network is parsed and used by the library. Figure B.2 in Appendix B clearly show a more complex structure than Figure B.3b. More examples of dependency graphs generated by the Network Analyzer can be found at Appendix B.

Summary

A framework like EDGECAFFE allows for a more flexible execution of DNNs. Additional tools and profiling modules let users optimize the execution for specific networks and specific target devices. The asynchronous nature of EDGECAFFE makes it possible the process multiple DNN inference requests at the same time. EDGECAFFE makes it possible to:

- Execute DNNs layer by layer. By only keeping the layers in memory that are currently executing, the memory usage can be reduced.
- Utilize multiple workers. Using multiple workers (threads) allows for the simultaneously preloading of future layers while the current layers is executed. This is only possible when the memory space allows for this.
- Profile networks. In order gather the required information of trained networks, such as memory usage, EDGECAFFE is able to profile existing trained networks.
- Multi Model execution. The fine-grained control by the layer by layer execution allows for multiple trained models to be executed simultaneously.

The EDGECAFFE implementation is open-sourced on Github[a].

[a] `https://github.com/bacox/edgecaffe`

6 Masa

Naively executing DNNs in the same way as is done in the cloud and on resourceful machines is not possible on edge devices without huge penalties. To overcome this a new algorithm is needed. In this chapter the Memory Aware Scheduling Algorithm (MASA) is proposed. MASA is the scheduling algorithm used for scheduling the tasks. This can only be done with a framework that offers partial execution of DNNs. EDGECAFFE is the framework that facilitates this and EDGECAFFE is explained in more detail in chapter 5.

6.1. Networks differ

Networks can vary greatly, in number of layers, in layer composition, in number of parameters and so on. In this section some differences between networks are discussed. Figure 6.1 shows the file-size distribution split by layer of each network in Table 3.1. The more layers of a certain size are in a model, the higher the density. The difference between networks are clearly visible. *SoS_GoogleNet* has a lot more small layers in terms of parameter file sizes than *ObjectNet*. These difference need to be taken into account when scheduling DNNs.

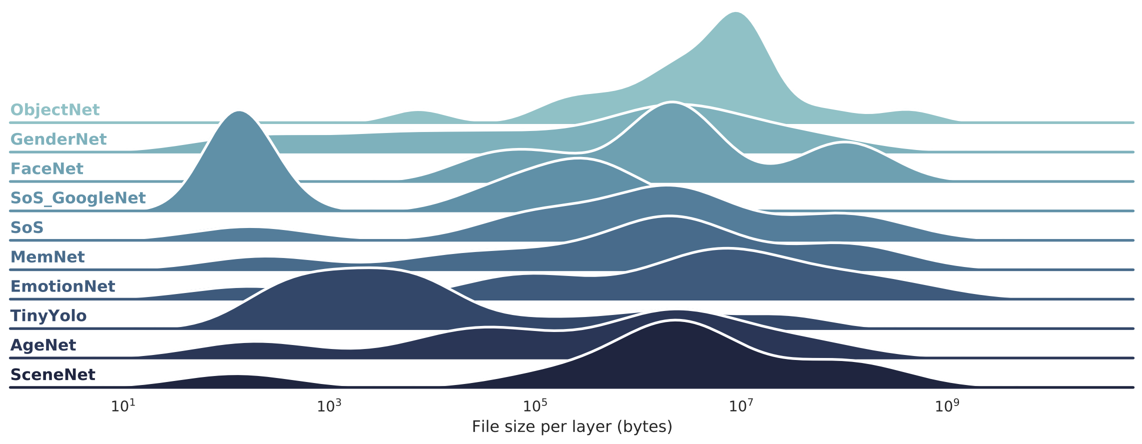


Figure 6.1: Parameter file size distribution (kernel density estimation) for different networks. The value of the y-axis shows the density of the distribution. The more layers of a certain size are in a model, the higher the density. It clearly shows that different networks have a different composition of layers and parameters file size.

6.1.1. Memory differs for networks and their layers

The runtime memory requirements of CNNs vary greatly. Traditionally DNNs are loaded in their entirety (bulk) before being executed [23, 50] favoring throughput over memory usage. Larger architectures with more and bigger layers require more memory. Table 6.1 shows examples of peak memory usage for different CNNs using Caffe (see section 4.1 for setup details). Peak memory usage ranges from 183 MB for AgeNet to 892 MB for SceneNet.

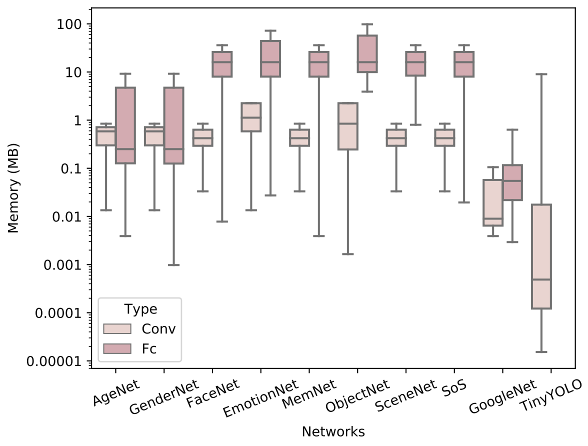


Figure 6.2: Peak memory distribution per layer type.

To lower peak memory requirements, especially to run large architectures on memory-constrained (edge) devices, we modify Caffe to allow fine-grained control on when layers are loaded and executed. However, memory usage can still significantly differ between layers. Figure 6.2 summarizes the per-layer peak memory distribution for all CNNs in Table 6.1. For each network and layer type the box shows the 25th, 50th, and 75th quartiles while the whiskers extend to the rest of the distribution. We skip pooling layers since they have negligible memory costs. For most networks, fully-connected layers have one order of magnitude higher memory usage. This is due to the high number of weights, which grows quadratically with the size of the layers, i.e., equal to the product of fully-connected and its input layer sizes. Convolutional layers only define a low number of weights since the filter matrix is shared across all receptive fields. A memory-aware scheduler needs to balance the memory usage with preloading layers for consistent performance.

Table 6.1: Overview of used DNNs in the experimental setup.

Network[1]	**Inference**	**Storage Space(MB)**	**Memory Usage(MB)**	**Layers** conv	fc	total
AgeNet [34]	Age	44	183	3	3	19
GenderNet [34]	Gender	44	186	3	3	19
FaceNet [14]	Face	217	875	5	3	23
SoS [73]	Saliency	218	875	5	3	21
GoogleNet [73]	Saliency	23	404	22	2	151
TinyYOLO [55]	Object	62	263	9	-	39
EmotionNet [35]	Emotion	378	761	5	3	22
MemNet [27]	Memorability	217	880	5	3	22
SceneNet [77]	Object	221	892	5	3	23

6.1.2. Memory matters - intra-network

The memory demands of modern CNNs can easily exceed the available memory, especially on resource-constrained edge devices, deteriorating inference responsiveness. Swapping allows OSs to execute programs with peak memory requirements that exceeds the available physical RAM by offloading memory pages to the swap space on disk. When a program accesses a memory page in the swap space a page fault is generated and the program execution is halted until the memory page has been restored to RAM. Since disk is orders of magnitude slower than RAM, programs can incur significant slowdowns based on the frequency of page faults. We demonstrate such an effect by bulk loading SceneNet as example. Figure 6.3 shows its mean execution time split per layer type under diminishing available memory. We repeat each experiment 10 times. Error bars report the standard deviation. The peak memory usage of SceneNet is 892 MB. If the available memory

[1]The architecture for SoS, GoogleNet, TinyYOLO, EmotionNet and MemNet are AlexNet, GoogleNet, Darknet, VGG-S and Hybrid-CNN respectively.

is above this value, i.e., 2 GB and 1 GB RAM, execution time remains unaffected. Lower values of available memory results in longer execution time as swapping kicks in. The higher the overcommitted memory the more page faults are generated, resulting in a higher execution speed penalty. With 256 MB RAM, the execution time is 7.2× slower compared to with 1 GB RAM. From the split layer statistics, one can see that the lower the memory requirements are the lower the slowdown is because the probability of a page fault is lower. With 512 MB RAM, the fully-connected layers are already affected significantly (2.5× slowdown), while the effect on convolution layers is (still) negligible. Overall this underlines how memory-awareness matters for good performance.

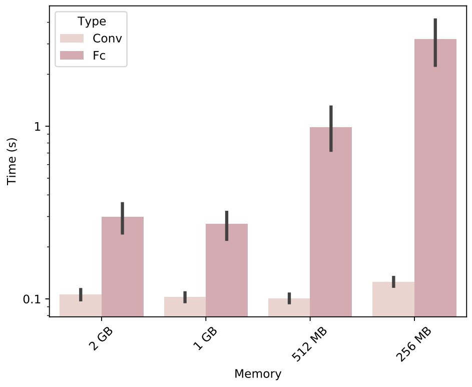


Figure 6.3: Impact of diminishing available memory

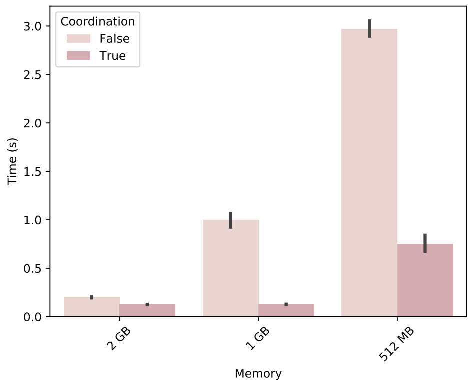


Figure 6.4: Impact of lack of multi-DNN coordination

6.1.3. Memory matters - inter-networks

Multi-DNN inference easily exacerbates the detrimental effects of insufficient memory. Each network increases memory usage. Without coordination, this increases the probability of page faults which negatively affect execution time. We exemplify this effect by running two instances of the same network (MemNet) to ease comparison. Each instance is pinned to a separate core, i.e., the instances share memory but no compute resources. We first run the two instances in parallel (without any coordination). Then we run one instance after the other (naive coordination). We repeat each experiment 10 times. Figure 6.4 compares the mean execution time of one MemNet instance across the two scenarios. Error bars indicate the standard deviation. Each MemNet instance uses 880 MB peak memory. When memory is sufficient, i.e., 2 GB RAM, both cases have similar execution times. The no-coordination case is slightly slower due to contention on other resources, i.e., mainly loading the weights from disk. However with multiple networks and no-coordination memory becomes a bottleneck already at 1 GB RAM. Here the no-coordination case is 7.8× slower. With 512 MB RAM, both cases are degraded but no-coordination is still 3.9× worse. Naive coordination does better but potentially misses out on complementary resource usages. This highlights the increased challenge and need for coordination when doing multi-DNN inference.

6.2. Common algorithms

There already exist numerous scheduling algorithms. All scheduling algorithms have limitations and drawbacks. In this section we consider a couple of well known scheduling algorithm for the DNN task scheduling problem. An overview of the algorithms can be seen in Table 6.2.

6.2.1. Round Robin

The Round Robin algorithms attempts to give each task (or process) an equal unit of time called a quantum. The focus of Round Robin is fairness. This means that tasks are preempted in due time to share the processing time with other processes. This behavior is undesirable when scheduling DNNs. The resource dependencies between the tasks of a DNN ensure penalties when using Round Robin in this situation. As the tasks are treated 'fair' generating major page fault is inevitable since all the loading tasks will overlap with each other.

When a task A of a DNN is committed to be executed, it is in the best interest of the rest of the tasks that the memory allocated by task A can be freed as soon as possible. It follows that a preemptive algorithm is counterproductive to this end.

Table 6.2: Properties of different scheduling algorithms.

	Round Robin	FcFs	SJF
Decision method	Preemptive	Non Preemptive	Non Preemptive
Response time	Low	High	Low for short tasks
Waiting time	Small	High	Low for short tasks
Throughput	Less for small quantum	High	High
Starvation	No	No	Possible

6.2.2. FcFs
The First Come Fist Serve (FcFs) is one of the most simple algorithms. The queue that holds the tasks is being processed from start to end. The tasks in the queue are 'ordered' based on the time that they are submitted to the queue. This method is agnostic to any other metric than arrival sequence. When running in single-worker mode, FcFs does not gain any potential speed up. When running this algorithm in a multi-worker setting, no speed up is gained since the dependencies lock the tasks in place.

6.2.3. Linear FcFs
When FcFs is combined with the Linear dependency graph we get the policy that uses the least memory possible. Every loading task is directly followed by its corresponding execution task, thereby retaining the memory as short as possible. The downside of this method is the fact that there are no speed ups possible by utilizing more than one worker. By using as little memory as possible it reduces the major page faults significantly. This means that Linear FcFs does not speed up the execution of DNN in any sense but it does not get slowed down by memory paging penalties.

6.2.4. SJF
The Shortest Job First (SJF) algorithm takes the job service time into consideration. This non-preemptive algorithm causes the average response time to decrease by giving priority to smaller tasks over larger tasks, where the size corresponds to the execution time. This unfair behavior is prone to starvation of large tasks especially in high workload environments. The SJF algorithm still ignores the resource requirements of the tasks and the given resource limitations of the device.

6.2.5. SNF
This variation of the Shortest Job First schedules accordingly to a larger granularity, networks instead of tasks. The Shortest Network First considers the execution time of the whole network (this is a composition of all the tasks in a network) as the sorting metric. This way memory is not retained as long as would be with SJF because the a network is finished a quickly as possible. The downside of this method is that is can lead to resource waste as we treat networks as a single block, as is shown in Figure 4.1.

6.3. Architecture Overview
As DNNs are used in two-fold, namely a training phase and a inference phase, the same goes for MASA. There are two stage: an offline stage and a online stage. In the former, trained networks are fed through a the Model Preparator (Figure 6.5). This component breaks apart the different model files into smaller layer files. Besides this, a more detailed model description is generated based on the trained model. The online stage is used for DNN inference. A prepared model is added to the Network Storage on a edge device for deployment. In the online stage, inference request can arrive from third party applications. Here MASA comes into play. When there are networks that needs to be processed, MASA tries to find a good schedule with the available resources in mind.

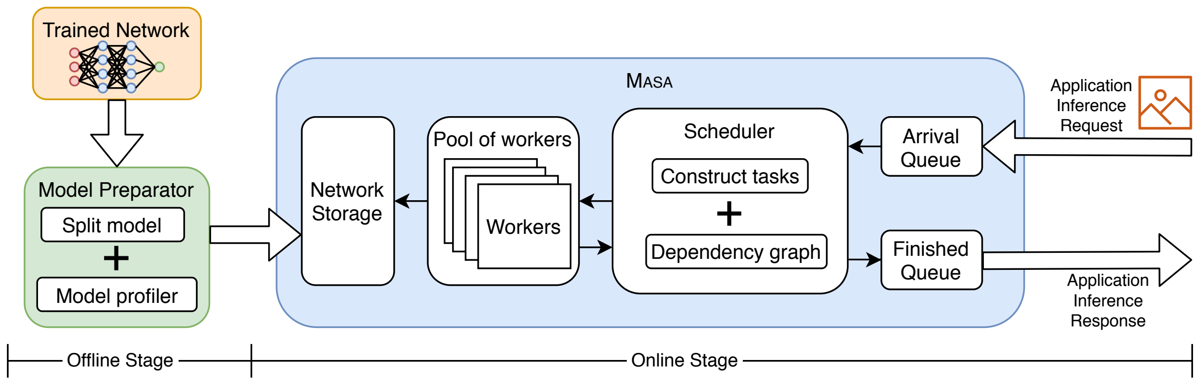


Figure 6.5: Architecture of MASA

6.3.1. Model Preparator

The Model Preparator is used to prepare models for edge inference. A trained model is composed of two elements: the model description and the model parameters. The description describes the structure ann the input dimension. The model parameters are the results of the training phase of the model. The Model Preparator uses the model description to split the network into smaller chunks according to the networks structure. The result is a set of smaller partial model parameter files that together compose the whole model. The trained models are offline prepared with inference on edge devices. After preparation, they are ready to be used on the edge devices.

6.3.2. Inference Engine

The inference en for computation without violating any dependency. Completed jobs are pushed to the finished queue.

This piece of software run on the edge device and is responsible to provide an inference result to an application the such a request is made. Third party applications can submit inference requests to MASA. An inference request is a tuple of an input (often an image) and a prepared model.

The general flow is the following:

1. A third party application makes an inference request.
2. The inference request is put into the arrival queue.
3. The scheduler pick the inference request out of the queue and generates the needed tasks according the model description.
4. The prepared model is retrieved from the Network Storage to be used in the generated tasks.
5. The scheduler generates a dependency graph based on the generated tasks.
6. The tasks are scheduled (if the dependencies allows for it).
7. When all the tasks of networks are executed, the network is finished and cleaned up.
8. The inference result is fed back to the third party application.

6.3.3. Scheduler

The scheduler sends the loading and execution tasks, in a non-preemptive manner, of all layers of DNNs that are requested for specific images to the workers. The scheduler follows two principles: (i) the task dependency, i.e., loading task of layer j should be completed before starting its execution task, and (ii) a hybrid order of layer type and memory constraint within and across multiple DNNs. As such, MASA incorporates the memory dependency at the inter- and intra-network levels.

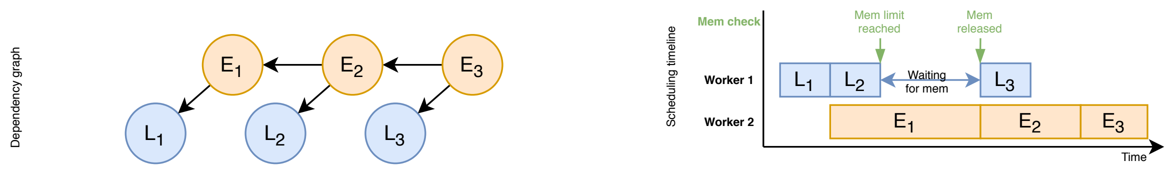
The loading and execution tasks of all layers across all specified DNNs are kept sorted into two groups: the waiting and ready group. Tasks start in the waiting group. Once all their dependencies are satisfied they are moved to the ready group. Dependencies are resolved based on the dependencies graphs defined by the

Algorithm 1: MASA scheduling algorithm

```
Function ValidTask(task):
    $c \leftarrow$ BusyWorkersCount()                         // busy worker count
    $M \leftarrow$ GetFreeMemorySpace()                       // available memory
    return $c == 0 \lor task.req\_memory \leq M$
Function ScheduleTask(w, tasks):
    for $task \in tasks$ do
        $M \leftarrow$ GetFreeMemorySpace()                   // available memory
        if ValidTask($task$) then
            $M = M - task.req\_memory$
            $w.Assign(task)$
            return True
    end
    return False
Procedure MASA()
    $W \leftarrow$ IdleWorkers()                              // set of idle workers
    for $w \in W$ do
        $exec, load \leftarrow$ ReadyTasks()                  // ready tasks by type
        if not ScheduleTask($w, exec$) then
            ScheduleTask($w, load$)
    end
    return
```

multi-DNN job and each DNN model. Tasks in the ready group are sorted to optimize memory consumption in a greedy fashion. First, execution tasks are prioritized over loading tasks since they free up memory upon completion. Second, tasks are sorted by increasing memory consumption. When on or more workers are available, tasks are pulled from the ready group in order, checking each time if their estimated required memory exceeds the available memory. If not, the task is started; otherwise the next task is checked. In case none of the ready tasks fits the available memory and all workers are idle, we forgo the memory constraint to allow one task to run to ensure progress. In this case we start the task with the smallest required memory of the preferred type, i.e., execution before loading. The pseudo-code of the scheduler is summarized in algorithm 1.

Figure 6.6 presents a scheduling example with a three-layer DNN and two workers for simplicity. The top part depicts the dependency graph between layers. The bottom part shows the scheduling timeline. Loading (blue, L) tasks must run before their corresponding execution (orange, E) tasks. Execution tasks must run in order, but loading task can be pre-loaded. At the start only L_1 can start since it does not depend on any other task. Once completed, both L_2 and E_1 are ready and executed one by each available worker. When L_2 completes the memory available is not sufficient to run neither E_2 nor L_3 until E_1 completes and frees up memory. At that point, with enough memory available, the two workers start L_3 and E_2 in parallel. Once E_2 completes we can start the last task E_3 and finish the inference.



(a) (b)

Figure 6.6: MASA scheduling a 3-layer DNN on 2 workers. Subfigure (a) shows the dependency graph. The scheduler introduces an idle period for worker 1 to prevent memory paging as is shows in subfigure (b).

6.4. Comparison with deterministic solution

The model presented in chapter 4 is able to provide the optimal schedule given the constraints of the model. This makes it possible the compare the schedule found by MASA against the schedule calculated by the CP-model. For this comparison, MASA is implemented in a discrete time simulator. This simulator simulates the execution of the tasks and the memory usage in a deterministic manner. The time steps taken by the simulator that is needed when executing MASA is compared to the length of the schedule found by the CP-model from chapter 4. From Figure 6.7 one can see the more workers are used in MASA the better is approximates the optimal schedule found by the CP-model. In the worst case, when the number of workers is 4, the schedule found by MASA is 22% longer than the schedule found by the CP-model. In the best case ,with 6 or 7 workers, the found schedule are of equal length.

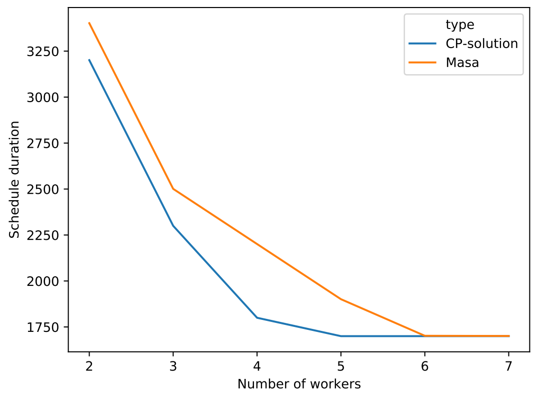


Figure 6.7: Multi model scheduling of two 4-layers networks. MASA approximates the optimal CP-schedule when the number of workers increases.

Summary

In this chapter we have shown an algorithm for processing DNN tasks. By making the scheduling algorithm resource aware, the paging penalties are reduced and the throughput of the system is increased. MASA is a non-preemptive, memory aware scheduling algorithm tailored for executing DNNs.

1. The dependencies that follow from the semantics of the DNN cause paging events when naive algorithms are used.
2. Resource aware algorithms reduce the number of paging events.
3. Existing scheduling algorithms like Round Robin, FcFs, and SJF are not directly suitable for this scheduling problem. The existing algorithms are prone to suffer from lower execution speed due to page faults.

7 Evaluations

This section presents our in-depth evaluation of MASA using real-world DNN applications on representative edge devices. Using extensive evaluation, roughly 35,000 inferences of trained DNNs, the performance of MASA is measured in relation to the `Bulk` and `DeepEye` algorithms. The full list of experiments can be found at Appendix C. All the results are summarized in Appendix D.

7.1. Experimental Setup

The testbed of EDGECAFFE from section section 5.4 provides some tools for a experimental setup (Figure 7.1). An experiment needs three elements: 1. A host system to run on, 2. A configuration file that describes the setup of both the host and EDGECAFFE, and 3. a workload that describes the arrivals and the timing and composition of these arrivals. The output of an experiment is the set of measurements that describe the performance of the system. For these experiments, we are not interested in the inference accuracy of the networks, but the timing results of the tasks and the usage of resources are most important. Since the models are structurally not changed, no model compressing, pruning or other intrusive changes, the model accuracy during inference is unchanged, hence the model accuracy need no measuring.

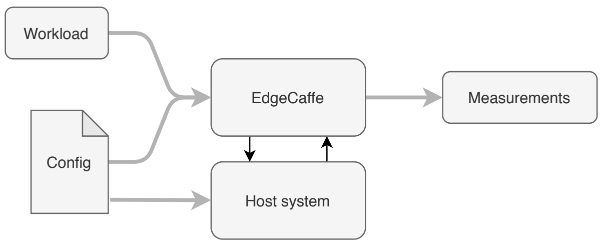


Figure 7.1: Experiment setup

The config file describes the configuration of both EDGECAFFE and the host system. For example the available resources for a given experiments are described in the config file as well as the scheduling algorithm to use.

The workload file describes the incoming inference requests. Each incoming request (arrival) consists of a arrival time, a set of networks to run, and the input data. The time describes the inter-arrival time between requests. When the time of all arrivals is set to 0, the workload represents a batch workload. In this case an arrival is submitted to system when the arrival in front of it has finished processing in EDGECAFFE. When the workload is not operating in batch mode, the arrivals will arrive at the system independent of each other, even when other networks are still being processed in EDGECAFFE. The workloads are generated offline to match the experiment and the target device. This is explained in more detail in section subsection 7.2.1 and subsection 7.3.1.

A host system can be used in two ways. Either all the available resources are allowed to be used by EDGE-CAFFE of the resources are limited by software. The former method is used when running on a target device

Table 7.1: Hardware configuration of the Raspberry Pi's used in the experiments.

	Model		
	RPi 3 Model B+	RPi 4 Model B	RPi 4 Model B
CPU	1.4 GHz	1.5 GHz	1.5 GHz
RAM	1 GB	2 GB	4 GB
GPU	VideoCore IV 250 MHz	VideoCore IV 500 MHz	VideoCore IV 500 MHz

like a Raspberry Pi that matches the setup of the experiment configuration. An example would be when running an experiment with 1 GB of RAM on a Raspberry Pi with 1 GB. The latter method is used when the host system has too much resources in comparison to the experiment description in the config file. An example of this method is when running experiments on a laptop, since laptops have on average more resource available than edge devices.

7.1.1. Resource limited environments

There are two ways to test EDGECAFFE and MASA in resource constrained environments. The first option is actually to use a limited device like a RaspberryPi. The second option is to limited the resources by software. As the testbed runs on Linux, tools are available to this end. The feature CGroups[1] allows for the isolation of the resource usage of a group of processes. This way EDGECAFFE can operate with subset of the resources that are available on the host system.

Listing 7.1: Restricting the host system to 2 GB of RAM using cgroups

```
# Create new cgroup
sudo cgcreate -g memory:force-swap
# Set virtual memory limit to 2 GB
sudo cgset -r memory.limit_in_bytes=2G force-swap
# Set allowed swapped memory to 8GB
sudo cgset -r memory.memsw.limit_in_bytes=8G force-swap
# Show the configuration
sudo cgget -g memory:force-swap | grep bytes

# Run process within cgroup
sudo cgexec -g memory:force-swap <program> <program arguments>

# Delete group if needed
sudo cgdelete -g memory:force-swap
```

Using a configuration like Listing 7.1, EDGECAFFE has 2GB of RAM available and 8GB of swap space. This allows for large models still to be executed but take penalty as using swap space is a order of magnitude slower than RAM.

7.1.2. Hardware

For edge devices we consider Raspberry Pi (termed RPi) as representative edge devices because of its wide adaptability and ease of programming. Specifically, we select three configurations of RPi: (i) RPi 3B+ equipped with Cortex-A53 (1.4 GHz) and 1 GB memory, (ii) RPi 4B with Cortex-A72 (1.5 GHz) and 2 GB memory, and (iii) RPi 4B with Cortex-A72 (1.5 GHz) and 4 GB memory. The hardware specification of the used Raspberry Pi's are listed in Table 7.1. To emulate the scenario of multiple applications on edge devices, we only consider memory sizes of 512 MB, 1 GB and 2 GB in the following evaluations. Each RPi is equipped with a SanDisk Extreme 64 GB microSD card for storage.

[1]https://man7.org/linux/man-pages/man7/cgroups.7.html

7.1.3. Available networks

We consider 9 types of DNNs, namely AgeNet, GenderNet, FaceNet, SoS, GoogleNet, TinyYOLO, EmotionNet, MemNet, and SceneNet, to analyze images from the EDUB-Seg dataset [11, 62]. We note that as the models are not altered, e.g., not compressed or pruned, the accuracy of the networks is not impacted. Each network conducts various kinds of image analysis, e.g., inferring age, gender, faces, and salient objects, and differ in their network structures and sizes. We summarize all networks considered in Table 6.1, where disk space and active memory usage obtained from our profiler are listed. EmotionNet is the largest network in terms of absolute storage space, i.e., in the order of 380 MB. SceneNet is the largest network in terms of memory usage, i.e., in the order of 890 MB.

We emulate scenarios of multi-DNN inference by executing multiple of such DNN models on periodically and stochastically generated image arrivals. The specific composition and number of DNN models are given in each subsection. For periodical arrivals, single images arrive at a fixed interval. For stochastic arrivals both inter-arrival times and number of DNNs per image follow normal and uniform distribution, respectively.

7.1.4. Performance measures

We aim to optimize the average response times per multi-DNN job. The response time is measured from the moment the image arrives till the time all DNN models have completed execution. For periodical arrivals, we also set deadlines which are imperative for safety critical systems. All values presented in the following sections are averaged over 150 images and more than 450 DNN inferences.

7.1.5. Dataset

The data used during the evaluations originates from the EDUB-Seg dataset [11, 62]. The data from this set is captured using wearable devices. The images are captured with the Narrative Clip [1] every 30 seconds. The dataset contains 18k egocentric images shot over a period of 20 days. This is ideal to imitate a lifelogging scenario.

7.1.6. Experiment naming scheme

In order to easily distinguish between experiments a naming scheme is used. An experiment is coded by using six variables. Using the following naming scheme experiments are identified: $T/N/M/D/W/I$. The values that the variable can take are the following:

- **T:** Type can be one of $\{S, B\}$ where S and B stands for *stochastic* and *batch* respectively.
- **N:** Name can a string, for example *lifelogging*
- **M:** Available memory can be one of $\{512M, 1G, 2G\}$
- **D:** Device can be one of $\{R1, R2, R4, L\}$ where $R1, R2, R4$ stand for RaspberryPi 1, 2, or 4 and L stands for *laptop*
- **W:** Number of workers where $W \in \mathbb{N}^+$
- **I:** Inter-Arrival Time modifier, only applicable for *stochastic* experiments, can be one of $\{0.8, 1, 1.2\}$

When data is combined, asterisks (*) can be used to indicate all values of a categorical variable.

A full list of all the experiments is located at Appendix C.

7.2. Batch Arrivals

When inference requests are bundled together it is called a batch arrival.This can happen when a single DNN does not provide enough information about the input. In that case multiple DNNs can be used on the same image to provide additional information. An example of this is when we want to know the location of the faces in an image as well as the gender and the age of the persons. When considering batch arrivals we assume that the networks used in a batch can work independent of each other on the input data. This allows for the maximum amount of parallelization.

7.2.1. Setup

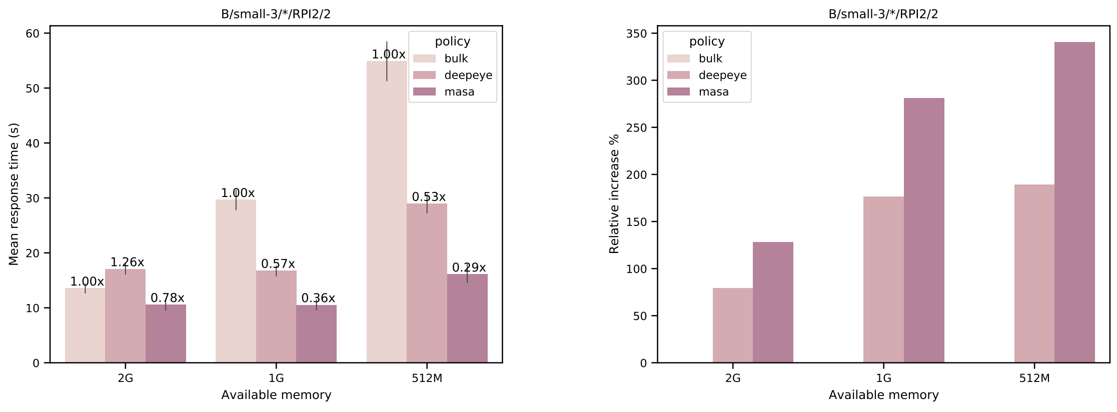
We consider five types of workload scenarios that analyze periodically images: (i) three small DNN models: AgeNet, GenderNet and TinyYOLO, (ii) three mixed DNN models both small and large: AgeNet, EmotionNet, and FaceNet,(iii) three large DNN models: MemNet, EmotionNet, and ObjectNet, (iv) lifelogging scenario where 5 DNNs are used to automatically annotate captured images: TinyYOLO, EmotionNet, MemNet, SceneNet, and SoS, and finally (v) all networks from Table 6.1. The DNN models used in the lifelogging scenario are typical networks used to give useful annotations to images captured during a lifelogging activity [42]. The images arrive every 20, 200, 200, 500 seconds for small, mixed, lifelogging, and large scenarios, respectively, and the analysis of multi-DNN needs to be completed before the arrival of the next job.

7.2.2. Results

We summarize the five scenarios in Figure 7.2, Figure 7.3, Figure 7.4, Figure 7.5, Figure 7.6. In all cases MASA outperforms the algorithms `Bulk` and `DeepEye`. For small and mixed networks, MASA achieves the lowest response time, trailed by `Bulk` and `DeepEye` across all combinations of workload scenarios and RPi configurations, as shown in Figure 7.2 and Figure 7.3. The performance of `DeepEye` often has the highest response times because multiple DNNs are greedily loaded into the memory and cause costly memory swap operations. Meanwhile, due to the extra consideration of intra-network dependency, MASA outperforms `Bulk` by at least 30 %, in case of 512 MB memory.

Another observation worth mentioning is the trend of performance gain of MASA. It has significant performance gains on devices with smaller memory, i.e., 512 MB, whereas the performance gain on 2 GB memory is less significant. Moreover, the relative performance gains of MASA against other approaches is the highest on small devices and homogeneous DNNs, compared to large devices and mixed workloads. This can be explained by the balanced task times of execution and loading and avoidance of memory swapping.

In Figure 7.5, we summarize the average response time of the lifelogging application that emulates a real-life application, executing five DNN inferences for every captured image. Similar trends as for the previous two scenarios can be observed: MASA can effectively allocate the limited worker threads and memory to achieve the lowest response time. The average response time of MASA in the case of 1 GB memory is similar to `Bulk` and `DeepEye` running on 2 GB memory. In other words, MASA can achieve almost 50% resource saving without degrading response times.



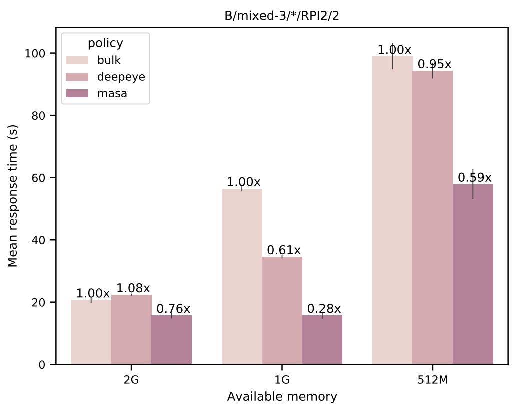
(a) Mean response time. Lower is better.

(b) Relative difference to the `Bulk` algorithm. Higher is better.

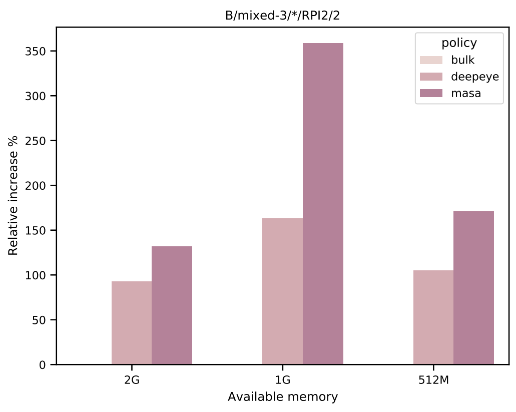
Figure 7.2: The result of the *small-3* Experiment. MASA outperforms `Bulk` and `DeepEye` in all memory spaces.

7.3. Stochastic Arrivals

On multi-tenant edge devices like mobile phone, multiple applications can request an inference at the same time or an inference request can arrive when there already is one running. With independent requesters, arrivals can come with a random intervals. In this section we investigate the effect of stochastic behavior when scheduling DNN inference requests.

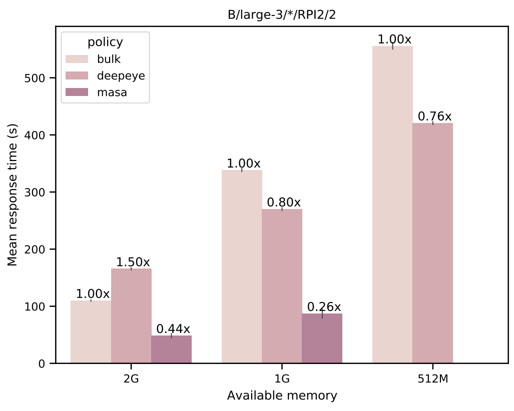


(a) Mean response time. Lower is better.

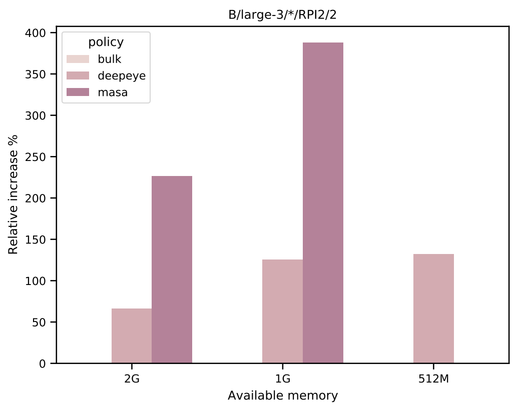


(b) Relative difference to the `Bulk` algorithm. Higher is better.

Figure 7.3: The result of the *mixed-3* Experiment.



(a) Mean response time. Lower is better.



(b) Relative difference to the `Bulk` algorithm. Higher is better.

Figure 7.4: The result of the *large-3* Experiment.

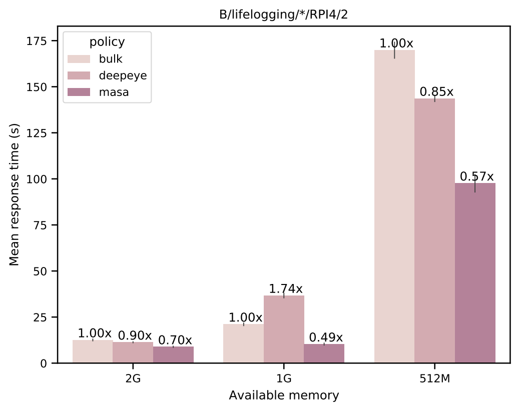
7.3.1. Setup

We consider three scenarios for stochastic behavior: (i) random arrivals with fixed composition, (ii) fixed arrivals with random composition, and (iii) the all-random scenario.

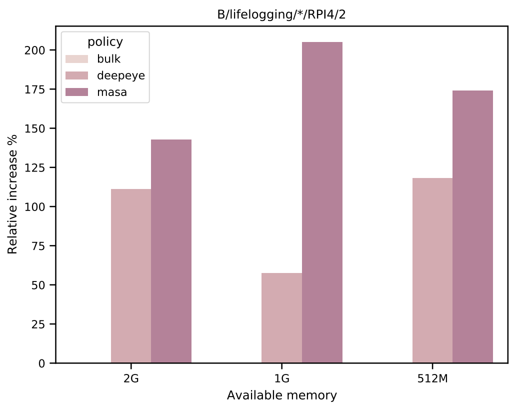
Here, we consider stochastic image analyses, where either the image arrivals or the number of DNN inferences, or both are generated stochastically. This workload scenario is much more complex than the periodic cases due to the intricate intra-network dependency and heterogeneity of inference jobs.

The arrivals of the images needs to be generated for each combination of scenario, and target device. This is because the inter-arrival time of each arrival is dependent on the execution time. Especially for the scenarios where the composition of networks can change every arrival. For every scenario, a list of arrivals is generated beforehand. First the combination of the arrivals (in terms of networks) is sampled. Based on the distribution of networks that appear in the arrival list, the mean inter-arrival time is calculated. Using this average inter-arrival time, the real inter-arrival times are sampled from a normal distribution where the mean is the aforementioned average inter-arrival time. This guarantees that on average the system has enough time to process the arrivals. Due to the randomness in the inter-arrival times and the random compositions of arrivals, it can happen that the system at specific points in time gets more networks than is can process and the inverse is true as well.

Specifically, three types of stochastic scenarios are evaluated with increasing stochasticity. (i) Scenario I: single DNN inference randomly drawn from the nine listed in Table 6.1 is requested upon the arrival of

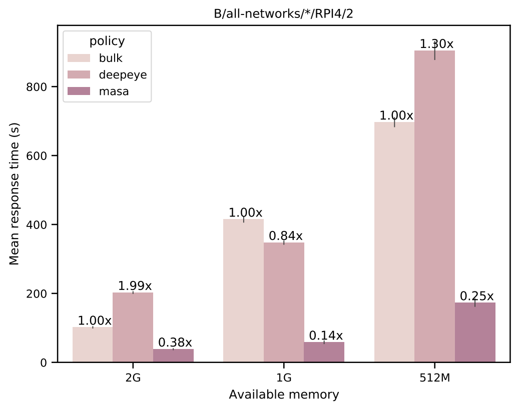


(a) Mean response time. Lower is better.

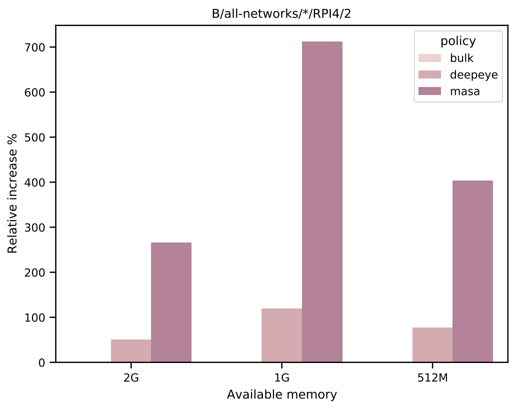


(b) Relative difference to the `Bulk` algorithm. Higher is better.

Figure 7.5: The result of the *lifelogging* Experiment.



(a) Mean response time. Lower is better.



(b) Relative difference to the `Bulk` algorithm. Higher is better.

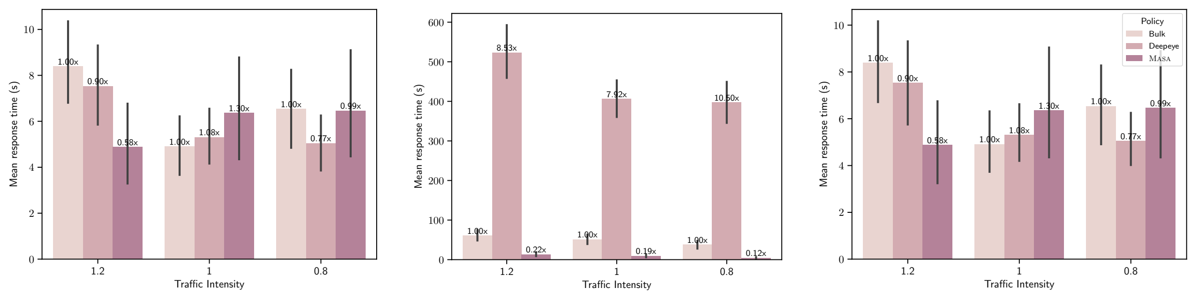
Figure 7.6: The result of the *all-networks* Experiment.

images following normal distribution where the mean is the sum of the average execution of the candidate models and a standard deviation of 200 ms, respectively. Due to the high variance of inter-arrival times, multiple DNNs need to be processed at the same time. (ii) Scenario II: the images arrive periodically but the composition of multiple DNNs is randomly requested following a uniform distribution. (iii) Scenario III: the images arrive according to a normal distribution and the composition of the networks is sampled randomly. For this scenarios we know as little information as can be. We don't know the interval and we don't know how many and what networks will arrive. For each scenario, we evaluate MASA on three memory configurations, and three traffic intensities. We normalize the traffic intensity with respect to the mean of the execution time of all networks in the scenario, as such values of 0.8, 1.0 and 1.2 are evaluated. The higher the value of traffic intensity becomes, the more challenging it is to achieve low response times.

7.3.2. Results

Figure 7.7 and Figure 7.8 summarize the average response times of the aforementioned stochastic scenarios and the relative performance gain respect to `Bulk` and `DeepEye`. Similar trends as the deterministic case can be observed: MASA achieves the lowest response times for small memory and high traffic intensity. In terms of absolute values, here the relative gains are significantly higher than for the deterministic scenarios which only need to cope with predictable workloads.

Scenario I. In Figure 7.7, one can clearly see that the performance gains of MASA against `Bulk` and `DeepEye` are more pronounced for higher traffic intensity and lower memory. In the cases of 1 GB and 512 MB memory (shown in Figure 7.7b and Figure 7.7c), we reduce the average response times up to 90%, compared to the second best policy. In the case of 2 GB memory shown in Figure 7.7a, we are slightly worse especially for the lighter traffic intensity. This can be explained by the fact that under light traffic greedy loading algorithms like `Bulk` and `DeepEye` are better to recover from the impact of memory paging than under heavy traffic. Moreover, we would like to point out that the average response times of `Bulk` and `DeepEye` increase from around 6 up to several hundreds seconds when memory space is reduced from 2 GB to 512 MB. In contrast, MASA can show only a 1.5X increase of the response times relative to the memory reduction, i.e., from 5 seconds at 2 GB up to 60 seconds at 512 MB.

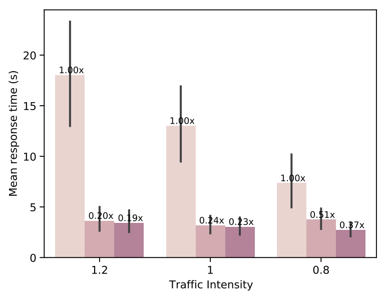


(a) 2 GB memory (b) 1 GB memory (c) 512 MB memory

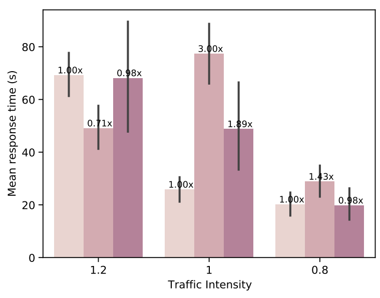
Figure 7.7: Response time of stochastic scenario I: comparisons of `Bulk`, `DeepEye` and MASA.

Scenario II. In Figure 7.8 we can see that MASA is still the best performing policy, trailed by `DeepEye` then `Bulk`. As this scenario is slightly more predictable than scenario I due to the periodical image arrivals, the performance gain is slightly lower than scenario I. `DeepEye` is able to manage the response time by greedily interleaving the model loading and execution. Here, MASA can maintain close to superlinear ratio between the average response times and available memory, i.e., from 3 seconds at 2 GB up to 30 seconds at 512 MB at a traffic intensity of 1.

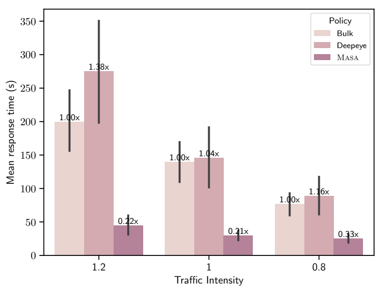
Scenario III In Figure 7.9, one can clearly see the impact of reduced memory on the performance of `Bulk`. While in Figure a MASA has only a 14% gain, in Figure 7.9b and Figure 7.9c the gain in performance of MASA over `Bulk` is significantly more. MASA is able to achieve such remarkable results due to its intelligent memory management and interleaved execution of DNN layers. We also evaluate the effectiveness of MASA on different number of worker threads. MASA can robustly ensure low response times, whereas `Bulk` and `DeepEye` can not properly take advantage of multiple worker threads. Increasing the number of workers to a value larger than 2 allows MASA to better interleave tasks when multiple DNNs are executed concurrently. This flexibility in scaling is not seen in `Bulk` and `DeepEye`.



(a) 2 GB memory

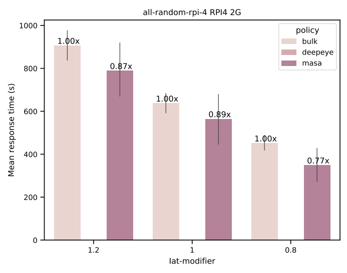


(b) 1 GB memory

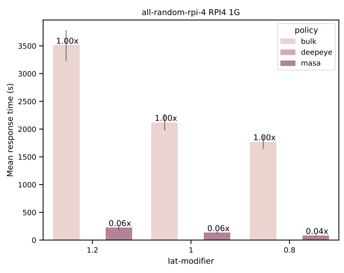


(c) 512 MB memory

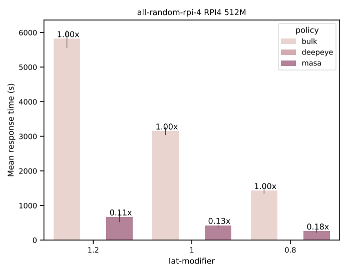
Figure 7.8: Response time of stochastic scenario II: comparisons of `Bulk`, `DeepEye` and MASA.



(a) 2 GB memory



(b) 1 GB memory



(c) 512 MB memory

Figure 7.9: Response time of stochastic scenario III: comparisons of `Bulk`, `DeepEye` and MASA.

7.4. Laptop results

Besides the Raspberry Pi's, we use a normal laptop for experimentation. The hardware specifications of the laptop are the following: Intel Core i7-6600U CPU 2.6 Ghz (4 cores). The operating version used on the device was the following: Ubuntu 18.04.4 where the version of the Linux kernel is 5.4.0-48-generic.

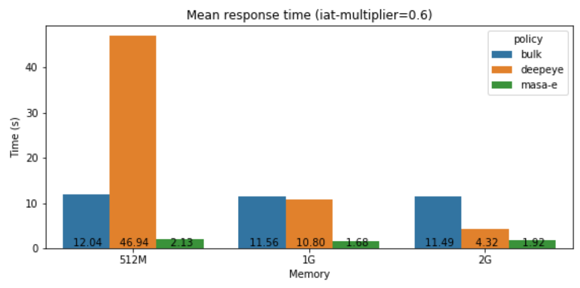
7.4.1. Four workers

To test the effect of the multi-threading in MASA experiments with four workers (threads) were executed. The first scenario considered for this was the triple AgeNet scenario. In the triple AgeNet scenario, the composition of networks in a single arrival consists of 3× AgeNet. The arrivals inter-arrival times are stochastic. For this scenario, the number of workers are increased from 2 to 4. This change causes the execution times of the networks to be different than during the experiments of section section 7.3. To counteract this change, the networks used in this four worker experiment were benchmarked again with the new configuration.

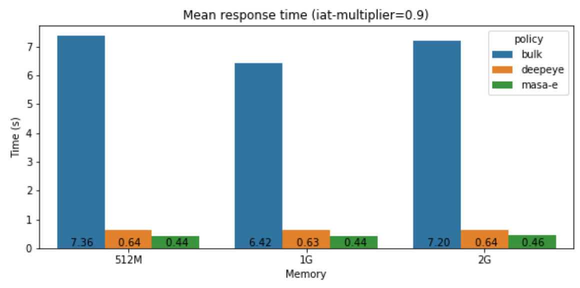
The results for the experiment with 2 workers is shown Figure 7.10. MASA performs better in all cases during this experiment. We can also clearly see that DeepEye has problems in very low memory spaces. From the DeepEye algorithm we know that the loading and execution speed do not take into account the current memory usage. Therefor there is a mismatch between the algorithm, memory space, and the set of networks that are queued. DeepEye is to greedy for the 512 MB environment in this case. To see the difference between the three algorithms Figure 7.11 show the relative difference in response time. The values of MASA are taken as baseline. It can be clearly seen that MASA outperforms Bulk and DeepEye with 30% to 80%.

When increasing the number of workers from 2 to 4, a similar trend can be seen in the results. Figure 7.12 shows that MASA outperforms Bulk and DeepEye. It for this experiment it becomes more interesting to compare the results of the 2 worker experiment and the 4 worker experiment. Comparing the values of Figure a with the values of Figure a show us the effect of doubling the number of workers. When the traffic intensity is low ($\hat{\rho} = 0.6$), MASA gains a 3.9× performance increase while Bulk and DeepEye only gain a 1.05× and a 1.17× performance increase respectively.

Looking at the low traffic intensity case ($\hat{\rho} = 0.6$), the gains for MASA are less impactful. MASA only gains a 1.09× performance increase while Bulk and DeepEye get a 1.17× and a 1.04× performance increase respectively.

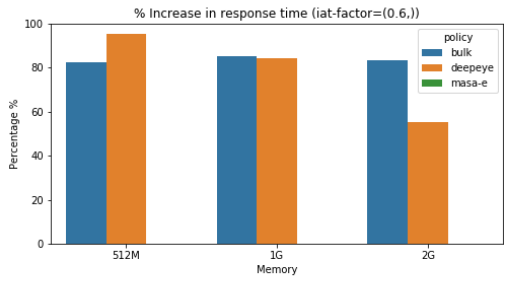


(a) Low traffic intensity scenario ($\hat{\rho} = 0.6$)

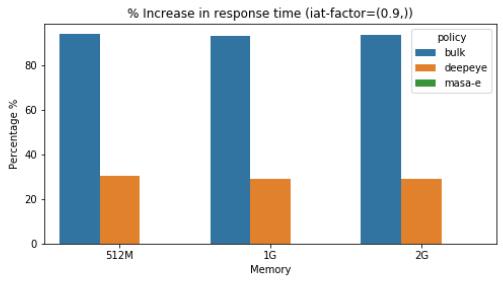


(b) High traffic intensity scenario ($\hat{\rho} = 0.9$)

Figure 7.10: Response times of the triple AgeNet 2 worker scenario: comparisons of `Bulk`, `DeepEye` and MASA. Missing values indicate that the testbed was not able to complete the experiment due to memory shortage.

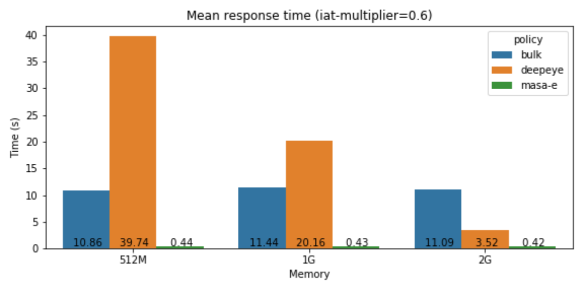


(a) Low traffic intensity scenario ($\hat{\rho} = 0.6$)

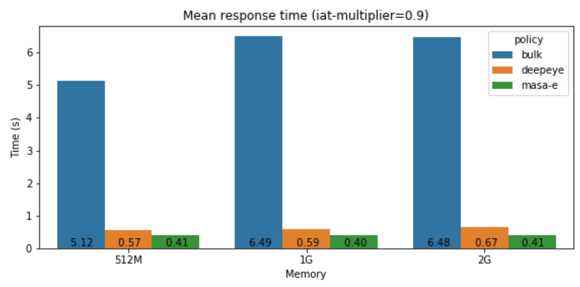


(b) High traffic intensity scenario ($\hat{\rho} = 0.9$)

Figure 7.11: The relative difference in response time of the triple AgeNet 2 worker scenario: comparisons of `Bulk`, `DeepEye` and MASA. The values of MASA are taken as baseline. Missing values indicate that the testbed was not able to complete the experiment due to memory shortage.

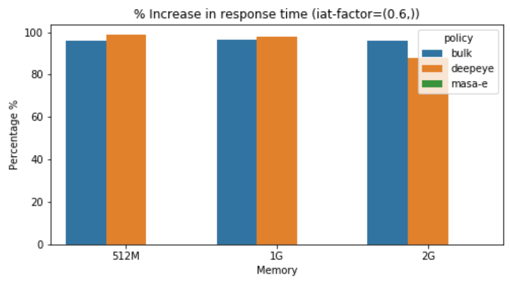


(a) Low traffic intensity scenario ($\hat{\rho} = 0.6$)

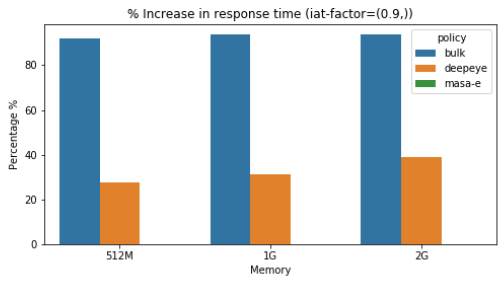


(b) High traffic intensity scenario ($\hat{\rho} = 0.9$)

Figure 7.12: Response times of the triple AgeNet 4 worker scenario: comparisons of `Bulk`, `DeepEye` and MASA. Missing values indicate that the testbed was not able to complete the experiment due to memory shortage.



(a) Low traffic intensity scenario ($\hat{\rho} = 0.6$)

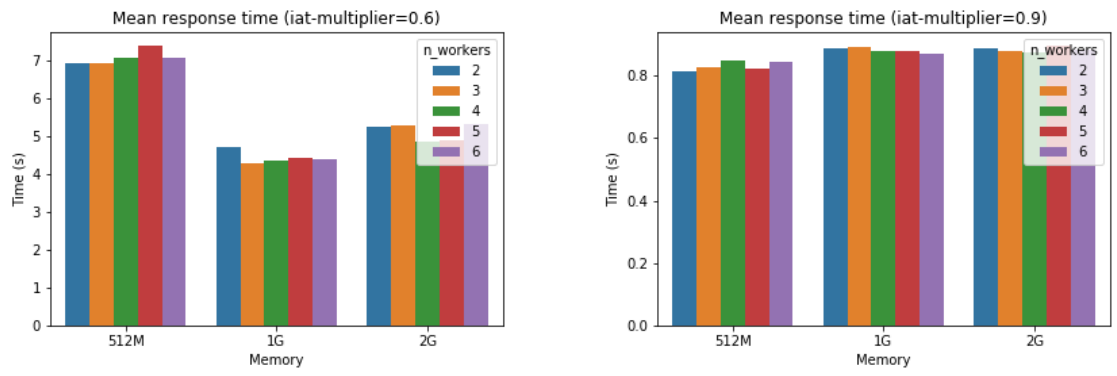


(b) High traffic intensity scenario ($\hat{\rho} = 0.9$)

Figure 7.13: The relative difference in response time of the triple AgeNet 4 worker scenario: comparisons of `Bulk`, `DeepEye` and MASA. The values of MASA are taken as baseline. Missing values indicate that the testbed was not able to complete the experiment due to memory shortage.

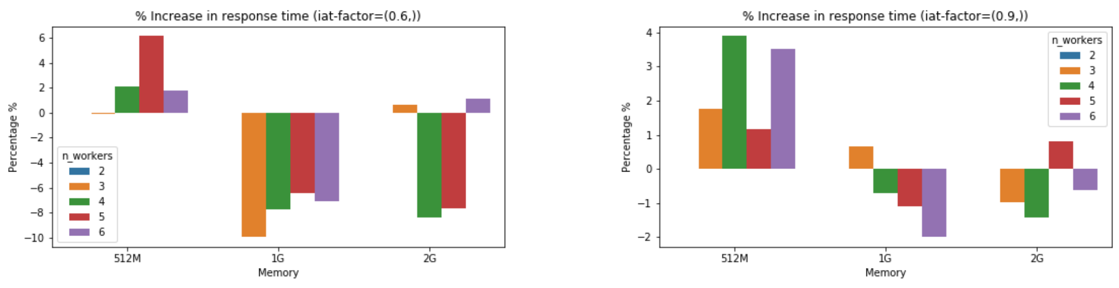
7.4.2. More workers

Besides only doubling the number of workers, increasing the number even further is the subject of this experiment. For this case only looked at Masa. The small network scenario is used where the number of workers is chosen van the range $[2,6]$. From Figure 7.15a and Figure 7.15b one can see that additional workers have little or a negative effect when operating in a low memory environment such as 512 MB. When more memory is available such as 1 GB and 2 GB, additional workers (< 2) can improve the response time. In this scenario the model (AgeNet) is relatively small which is beneficial. When using larger models, the performance gain will be less when using the same memory space. To reach a comparable performance (as in Figure 7.15) with larger models, a larger memory space is required than 2 GB.



(a) Low traffic intensity scenario ($\hat{\rho} = 0.6$) (b) High traffic intensity scenario ($\hat{\rho} = 0.9$)

Figure 7.14: Response times of the triple AgeNet 4 worker scenario: comparisons of `Bulk`, `DeepEye` and Masa. Missing values indicate that the testbed was not able to complete the experiment due to memory shortage.



(a) Low traffic intensity scenario ($\hat{\rho} = 0.6$) (b) High traffic intensity scenario ($\hat{\rho} = 0.9$)

Figure 7.15: The relative difference in response time of the triple AgeNet 4 worker scenario: comparisons of `Bulk`, `DeepEye` and Masa. The values of Masa are taken as baseline. Missing values indicate that the testbed was not able to complete the experiment due to memory shortage.

7.5. Overhead

When proposing a new method, it is important to investigate how much overhead is introduced. To this end we look at a couple of factors namely the I/O overhead and the memory overhead.

7.5.1. I/O

The parameters of a trained model are stored on disk (or any other secondary storage device) and needs to be loaded into memory before use. Since reading the contents of a file requires system-calls, we have to wait for the operating system.

A trained model is broken down by Masa into smaller pieces to give more flexibility in terms of scheduling. By splitting the model parameter file in smaller files additional I/O system-calls do need to be made to collect the whole model. When splitting a model in N pieces we introduce $N-1$ additional I/O system-calls. The question arises if these extra calls introduce a significant overhead in the system. When reading data from

Table 7.2: I/O overhead

	Model	Mean duration	std dev	mean sys-call time	% sys-call time
Single file	TinyYolo	233.883 ms	36.9966 ms	39.423084 ms	16.9%
Multiple files	TinyYolo	229.015 ms	41.2151 ms	39.159502 ms	17.1%
Single file	ObjectNet	3223.22 ms	79.6408 ms	438.7068 ms	13.6%
Multiple files	ObjectNet	3415.09 ms	98.487 ms	446.2022 ms	13.1%
Single file	SoS_GoogleNet	94.2531 ms	14.0528 ms	14.707980 ms	15.6%
Multiple files	SoS_GoogleNet	93.8436 ms	13.2665 ms	15.927964 ms	16.9%

disk to memory as system call is made by the program to the kernel to read a certain file. In Linux we can measure the time spent in a program waiting for system calls. Using the tool *strace*[2] all the systems calls made are captured. The experiment is setup as follows: A normal trained model is loaded into memory and the time that is takes as well as the used system calls are recorded. The same is done when reading a splitted model, all the individual partial files are read sequentially and the summed time and all the made system calls are recorded. The eliminate randomness as much as possible, this experiment is repeated 500 times. The results show that there is not an increase in time to load the whole model when we compare the loading of a full model and the loading of all the partial models. Looking at the data of the system calls captured with *strace*, there is no difference in the time spent waiting on system calls in both cases. The splitting of the model has not a significant effect on the loading time of parameters of a model.

In Table 7.2 can be seen that there is no significant difference between loading a model as a single file and loading as a set of smaller files. TinyYolo and SoS_GoogleNet are quicker when loaded as multiple files while ObjectNet is quicker when loading as a single file. Looking at the time that is spent in system-calls, there is not significant difference as well. When we normalize the system-call time (last column Table 7.2) and calculate the p-value we get 0.59. We can conclude that splitting a DNN model does not introduce a measurable overhead in terms of I/O system calls. The values for the I/O overhead were measured on the laptop of whose specifications can be found at section 7.4.

7.5.2. Memory footprint

By introducing a thread to control and coordinate the execution of DNNs, additional memory is used. Since memory is a scarce resource in edge devices and commonly a bottleneck, it is important to measure the memory footprint of the system. Using the tool `valgrind`, the memory footprint of EDGECAFFE was measured. While measuring, all the components were active, the worker threads are running but no DNN are loaded and processed. The system was kept in idle state for 10 seconds before shutting down. When no models were processed, when the system was in idle state, the used memory was 4.8 MB. When running in the smallest memory environment (512 MB) this accounts for less then 1% of the available memory.

7.6. Sensitivity Analysis

Here, we test the robustness of MASA in response to inaccurate memory estimates. To such an end, we multiply the profiled memory requirements of all layers in AgeNet by 0.5 and 0.75 to show under-estimation and by 1.25 and 1.5 to show over-estimation of memory. We run a batch of 6 AgeNet networks with a repetition of 20 image arrivals. We summarize the results in Figure 7.16. The under-estimated memory values cause the algorithm to load too many networks at the same time while the over-estimated memory values cause the algorithm to under-utilize the available memory. In the case of severely underestimation (the 0.5 multiplier), the average response times increase from 0.8 to 1.4 seconds as well as the standard deviation. Overall, conservative estimation of memory (even up to 50% higher than actual values) only incur small percentage increases of response times, e.g. 11.5% for 50% overestimation.

[2]https://man7.org/linux/man-pages/man1/strace.1.html

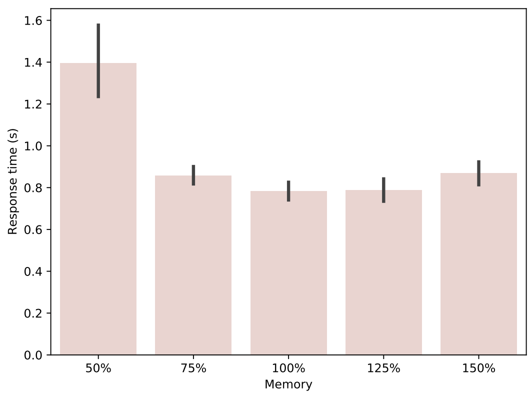


Figure 7.16: The effect of under- and over-estimation of the required memory for a network, where 50%, 75%, 125%, and 150% represents the percentage of memory values from the profiler.

Summary

We extensively evaluate MASA on two kinds of edge devices. A wide range of different scenarios shows that MASA is able to outperform `Bulk` and `DeepEye` consistently. The proposed solutions introduces minimal overhead in order to provide adaptability to (changing) resource constraints.

- MASA is up to 6× faster than Bulk and 1.5× faster than DeepEye when dealing with batch arrivals.
- MASA is up to 18× faster in encountering stochastic arrivals in comparison to Bulk and DeepEye.
- Increasing the number of workers has a positive effect on the performance of MASA when the memory space allows for it.
- Finding correct memory estimation for the tasks is important for efficient scheduling on edge devices.

8 Conclusion and future work

While Deep Learning is profoundly used in resource rich environments such as cloud systems and powerful desktops, edge devices are not yet utilized to great extend. Adaptability to (changing) resource limitations is a missing property in recent approaches of DNN edge inference. chapter 3 showed that model compression is a well used method. This however forces the model to adapt to resources by sacrificing model accuracy instead of using a resource adaptive execution method. A new DNN execution framework, EDGECAFFE, is presented in chapter 5 that offers partial execution, inter-network coordination, profiling of trained networks, and adaptability to resource limitations. This method of execution allows only the layers that are currently executed to be loaded in memory, thereby reducing the memory footprint of the models up to 88% and on average 35%. In order to exploit the partial execution, MASA is introduced in chapter 6. This memory aware algorithms is able to adapt to the availability of the resources. Extensive evaluation in chapter 7 shows that MASA is able to outperform `Bulk` and `DeepEye`. In both batch (section 7.2) and stochastic scenarios (section 7.3), MASA has a performance gain up to 8× and 16× respectively. At the beginning of this work, five research questions were presented. The chapters preceding the conclusion provide answers to these questions.

RQ1. *How can DNNs be decomposed in smaller components?*

Neural networks are a composition of layers. By examining the way a Deep Neural Network is executed on a machine, as is done in section 4.1, the different types of actions (*load, execute,unload*) a network can take can be derived. As the neural networks are a composition of layers, these actions are equal for each individual layer. By defining the actions a layer can take, a network can be described as a set of related tasks. Neural networks can be described in smaller tasks (*load, execute,unload*) on the granular level of the layers.

RQ2. *How can the execution of a DNN be speed up using the multiple CPU cores?*

Following the semantics of Neural Networks partial orders exists between certain tasks (subsection 4.1.2). These partial orders constraint the degrees of freedom between the tasks during determining the order of execution. By creating a dependency graph of a networks, as is described in Figure 4.1.2, parallelization of execution of tasks can be determined. Loading tasks of the same network are independent of each other, there are no dependencies between loading tasks. Execution tasks of different networks are also independent to each other. Execution tasks of the same network must be executed in the correct partial order, there exists dependencies between execution tasks of the same network. There are two obvious ways of speeding up the execution of Deep Neural Network. The first one the utilizing the lack of dependencies between loading tasks. With multiple core, we can pre-load layer $i+1$ while simultaneous execute layer i. The second method to speed up the execution of Neural Network is by running multiple networks at the same time. When the memory space allows for it, a second network can start executing while the first if already running.

RQ3. *What are the resource needs of a DNN?*

From section 2.1 and from section 4.1 we know how DNNs work internally and how they are executed. From this information it is derived that the main resource requirements of neural networks are computational effort in terms of CPU time and a certain amount of memory. From section 6.1 we know that networks behave very different depending on their composition. By describing the execution of a network in three tasks (*load, execute,unload*) we have a way to model the needs of a network (or layer).

> **RQ4.** *What scheduling algorithm can be used to schedule multiple DNNs on resource constrained devices?*

An important property the scheduling algorithm is resource awareness. From results in subsection 6.1.2 we can see that the penalty for exceeding the available resource limit can be very severe, resulting in significant slow down of execution speed or even resulting in a crash of the system. Well known existing scheduling algorithms like *First Come First Serve* and *Shortest Job First* are not able to take into account the dependencies of the task and are ignorant towards the resource limits (section 6.2). DeepEye, an algorithm with the focus on the execution of multiple Deep Neural Networks at the same time is not able to respect the resource limit when the traffic intensity is higher than expected as is shown in section 7.3. In order to schedule DNNs, an algorithm needs to be resource aware, and utilize the existing relationships between the tasks. MASA is proposed to combat the limitation of the aforementioned algorithms. With rigorous evaluations we show that MASA responds relatively well to imperfect information (due to benchmarking inaccuracies), to changing traffic intensities, and to different compositions of networks (section 7.2 and section 7.3).

> **RQ5.** *How can multi-model DNN inference scale on different devices with different resources?*

By using resource aware scheduling algorithms like MASA, DNN inference becomes more adaptive towards change. This means that even when parameters change (for example the available memory), DNN inference can still be executed in an efficient manner. With thorough evaluations, MASA has shown to outperform the existing solutions for multi model DNN inference (section 7.2). While `DeepEye`, due to a fixed execution schema, has difficulty with increasing the number of threads, MASA can easily scale up the number of threads. When increasing the number of threads subsection 7.4.1, we see that the performance gain is better compared to the performance gain of existing algorithms.

8.1. Future work

During this research, the focus was fully on the inference part of Deep Neural Networks. The process of training model is more challenging due the fact that besides the forward pass, a backward pass must be executed to correct the parameters. Although the performance gain is likely less when using MASA for training than it would be for inference, gains can still be made. One line of future research can be in exploring how partial execution provided by EDGECAFFE and a resource aware algorithms like MASA can increase the performance of training Deep Neural Networks on small devices with limited resources (edge devices).

Second, the diversity of the DNN models can be expanded. Research in the field of Deep Learning moves fast and new models are available with an ever increasing pace. Future research can look the different types of models and how their structure can be used of acceleration of the inference phase. Networks like Recurrent Neural Networks, Variational Auto-encoders, and the more recent transformer and attention based network offer a new challenge due to their more complex structure and more elaborate task dependencies.

As the third point, this research restricted to inference in CPU only. Looking into the effect of using GPUs and other hardware accelerators like digital signal processors (DSP) offer new challenges that can be worthwhile of exploring.

A
Constraint Programming model

A.1. Domain

- T = set of tasks
- W = set of workers
- S_i = Starting time of task i
- P_i = Processing time of task i
- C_i = Completion time of task i
- $X_{i,j}$ = decision variable if Task i is scheduled on worker j
- $\phi(A,B)$ = if task A is dependent on task B
- M = (Global) available memory capacity in the system
- m_i is task specific memory usage of task i
- τ_l = specific moment at time step l
- $\theta(A,B)$ = if Task A requires the memory of Task B
- $D_{d,e}$ = start time of locked memory interval from task d to task e
- $E_{d,e}$ = end time of locked memory interval from task d to task e
- $I_{d,e}$ = memory usage of of locked memory interval from task d to task e

A.2. Optimization function

$$\min(C_{\max}) \tag{A.1}$$

A.3. Constraints

$$\sum_{w \in W} X_{i,w} = 1,\ i \in T \tag{A.2}$$

$$C_i = S_i + P_i,\ i \in T \tag{A.3}$$

$$P_i \geq 0, \forall i \in T \tag{A.4}$$

$$[S_i, C_i] \cap [S_k, C_k] = \emptyset,\ \forall i, k \in T : i \neq k \text{ and } X_{i,w} = X_{k,w} = 1, w \in W \tag{A.5}$$

$$C_i \le S_k,\ \forall i, k \in T : \phi(k, i) = 1 \tag{A.6}$$

$$M_{tasks} = \sum_{i \in T} (S_i \le \tau_l \wedge \tau_l \le C_i) * m_i,\ \forall l \in \mathbb{N} \tag{A.7}$$

$$M_{locked} = \sum_{d,e \in T : \theta(e,d)} (D_{d,e} \le \tau_l \wedge \tau_l \le E_{d,e}) * I_{d,e},\ \forall l \in \mathbb{N} \tag{A.8}$$

$$M_{system} \ge M_{tasks} + M_{locked} \tag{A.9}$$

B
Dependency Graphs

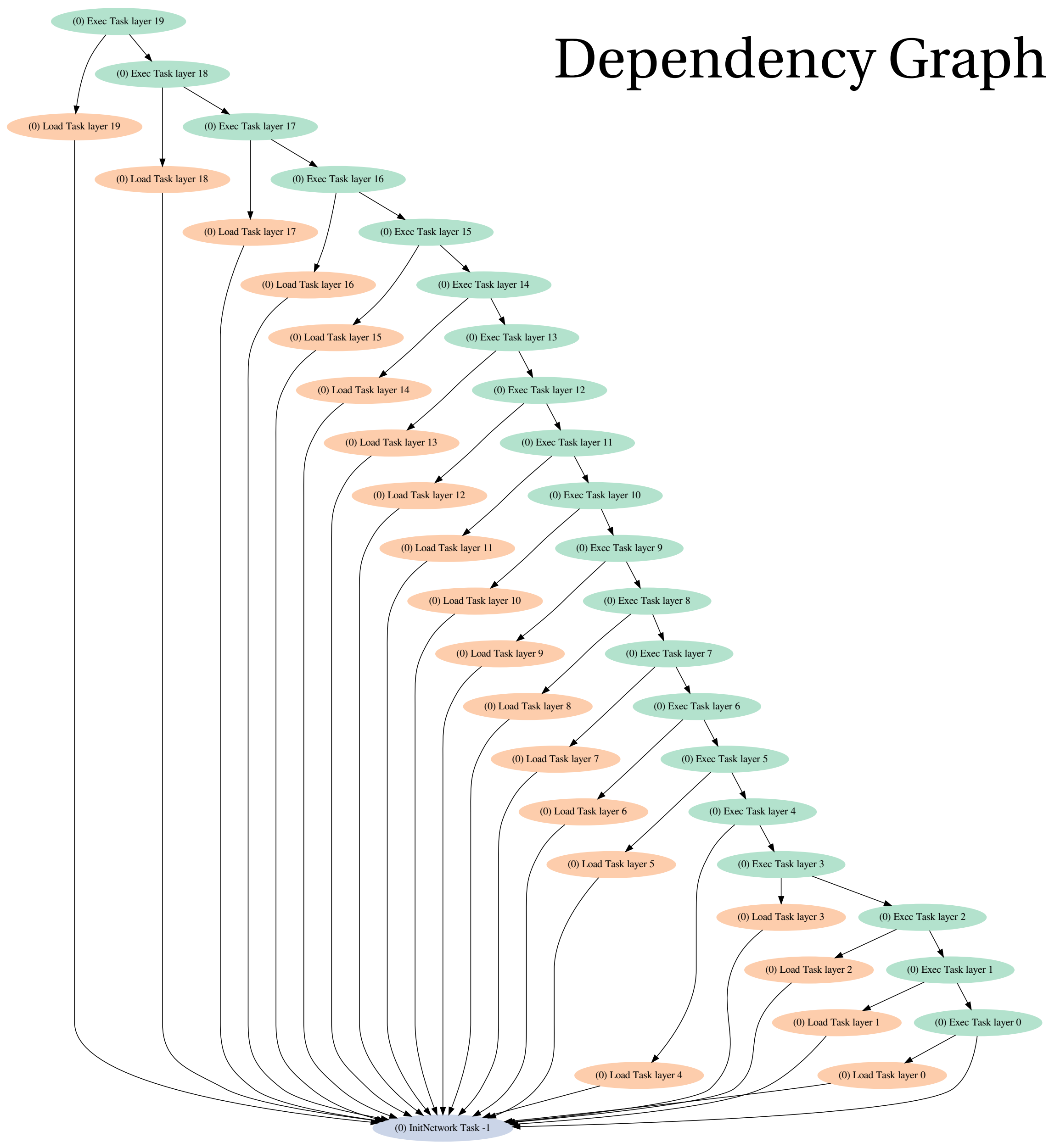


Figure B.1: Dependency graphs for the AgeNet network visualized for the policy masa.

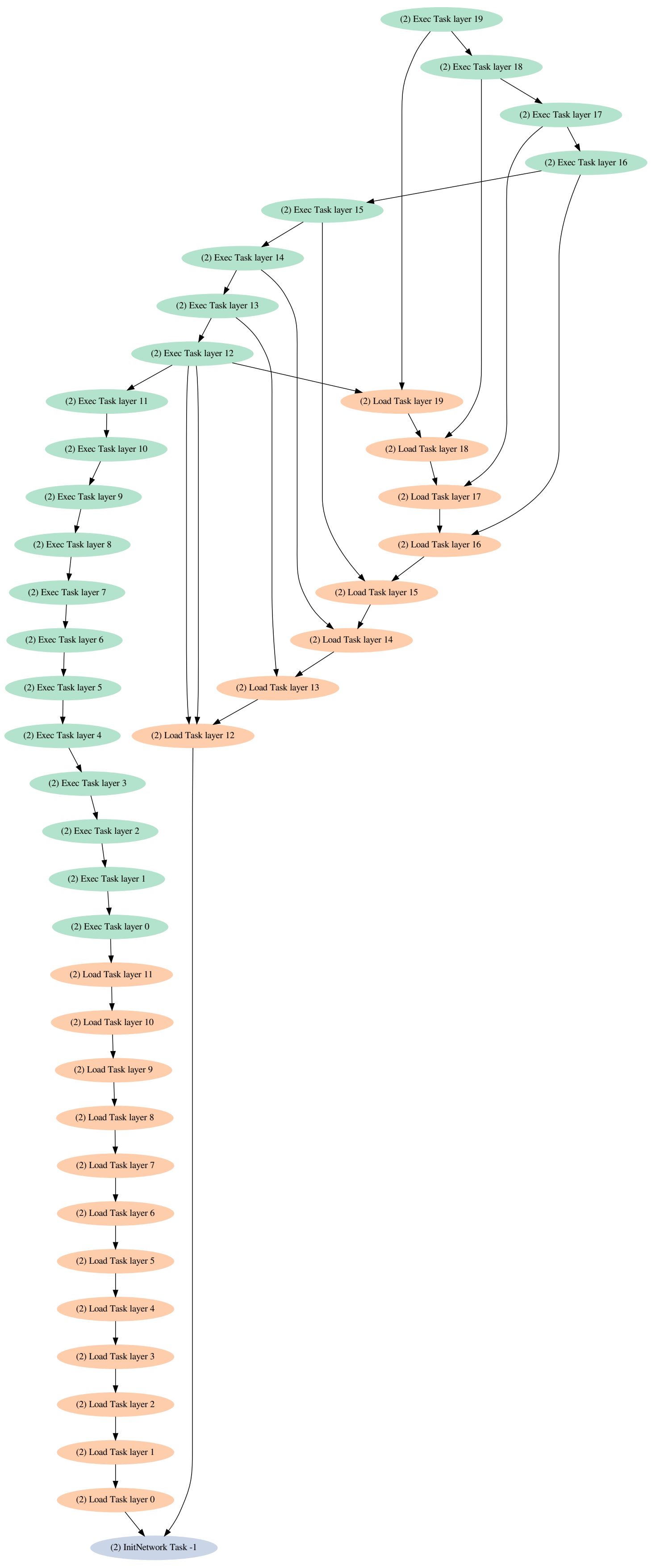
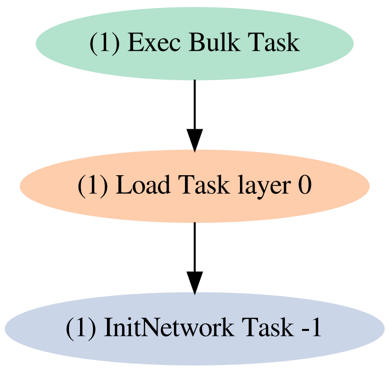
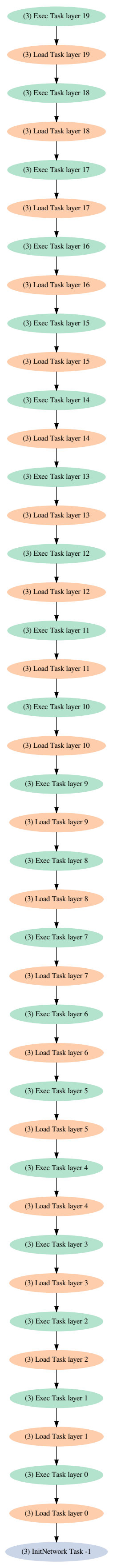


Figure B.2: Dependency graphs for the AgeNet network visualized for the policy deepeye.



(a)



(b)

Figure B.3: Dependency graphs for the AgeNet network visualized for the policies bulk and linear.

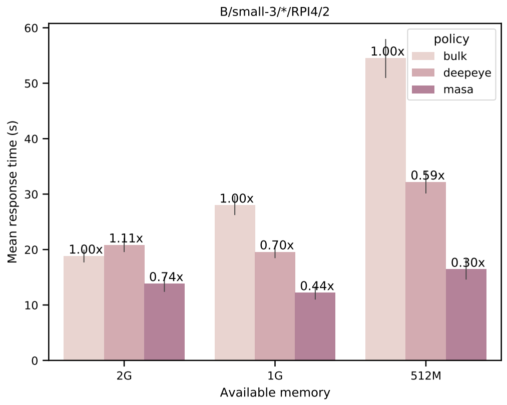
C
Overview of experiments

Table C.1: Overview of experiments

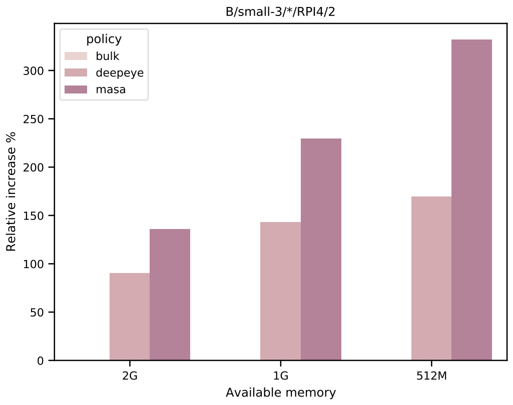
Code	Type	Name	Memory	Device	Workers	IAT
B/small-3/512M/R4/2	Batch	small-3	512M	RPI-4	2	-
B/small-3/1G/R4/2	Batch	small-3	1G	RPI-4	2	-
B/small-3/2G/R4/2	Batch	small-3	2G	RPI-4	2	-
B/mixed-3/512M/R4/2	Batch	mixed-3	512M	RPI-4	2	-
B/mixed-3/1G/R4/2	Batch	mixed-3	1G	RPI-4	2	-
B/mixed-3/2G/R4/2	Batch	mixed-3	2G	RPI-4	2	-
B/large-3/512M/R4/2	Batch	large-3	512M	RPI-4	2	-
B/large-3/1G/R4/2	Batch	large-3	1G	RPI-4	2	-
B/large-3/2G/R4/2	Batch	large-3	2G	RPI-4	2	-
B/lifelogging/512M/R4/2	Batch	lifelogging	512M	RPI-4	2	-
B/lifelogging/1G/R4/2	Batch	lifelogging	1G	RPI-4	2	-
B/lifelogging/2G/R4/2	Batch	lifelogging	2G	RPI-4	2	-
B/all-networks/512M/R4/2	Batch	all-networks	512M	RPI-4	2	-
B/all-networks/1G/R4/2	Batch	all-networks	1G	RPI-4	2	-
B/all-networks/2G/R4/2	Batch	all-networks	2G	RPI-4	2	-
B/small-3/512M/R2/2	Batch	small-3	512M	RPI-2	2	-
B/small-3/1G/R2/2	Batch	small-3	1G	RPI-2	2	-
B/small-3/2G/R2/2	Batch	small-3	2G	RPI-2	2	-
B/mixed-3/512M/R2/2	Batch	mixed-3	512M	RPI-2	2	-
B/mixed-3/1G/R2/2	Batch	mixed-3	1G	RPI-2	2	-
B/mixed-3/2G/R2/2	Batch	mixed-3	2G	RPI-2	2	-
B/large-3/512M/R2/2	Batch	large-3	512M	RPI-2	2	-
B/large-3/1G/R2/2	Batch	large-3	1G	RPI-2	2	-
B/large-3/2G/R2/2	Batch	large-3	2G	RPI-2	2	-
B/lifelogging/512M/R2/2	Batch	lifelogging	512M	RPI-2	2	-
B/lifelogging/1G/R2/2	Batch	lifelogging	1G	RPI-2	2	-
B/lifelogging/2G/R2/2	Batch	lifelogging	2G	RPI-2	2	-
B/all-networks/512M/R2/2	Batch	all-networks	512M	RPI-2	2	-
B/all-networks/1G/R2/2	Batch	all-networks	1G	RPI-2	2	-
B/all-networks/2G/R2/2	Batch	all-networks	2G	RPI-2	2	-
B/small-3/512M/R1/2	Batch	small-3	512M	RPI-1	2	-
B/small-3/1G/R1/2	Batch	small-3	1G	RPI-1	2	-
B/mixed-3/512M/R1/2	Batch	mixed-3	512M	RPI-1	2	-
B/mixed-3/1G/R1/2	Batch	mixed-3	1G	RPI-1	2	-
B/large-3/512M/R1/2	Batch	large-3	512M	RPI-1	2	-
B/large-3/1G/R1/2	Batch	large-3	1G	RPI-1	2	-
B/lifelogging/512M/R1/2	Batch	lifelogging	512M	RPI-1	2	-
B/lifelogging/1G/R1/2	Batch	lifelogging	1G	RPI-1	2	-
B/all-networks/512M/R1/2	Batch	all-networks	512M	RPI-1	2	-
B/all-networks/1G/R1/2	Batch	all-networks	1G	RPI-1	2	-
S/all-random/512M/R1/2/0.8	Stochastic	all-random	512M	RPI-1	2	0.8
S/all-random/1G/R1/2/0.8	Stochastic	all-random	1G	RPI-1	2	0.8
S/all-random/2G/R1/2/0.8	Stochastic	all-random	2G	RPI-1	2	0.8
S/all-random/512M/R1/2/1	Stochastic	all-random	512M	RPI-1	2	1
S/all-random/1G/R1/2/1	Stochastic	all-random	1G	RPI-1	2	1
S/all-random/2G/R1/2/1	Stochastic	all-random	2G	RPI-1	2	1
S/all-random/512M/R1/2/1.2	Stochastic	all-random	512M	RPI-1	2	1.2
S/all-random/1G/R1/2/1.2	Stochastic	all-random	1G	RPI-1	2	1.2
S/all-random/2G/R1/2/1.2	Stochastic	all-random	2G	RPI-1	2	1.2
S/random-composition/512M/R1/2/0.8	Stochastic	random-composition	512M	RPI-1	2	0.8
S/random-composition/1G/R1/2/0.8	Stochastic	random-composition	1G	RPI-1	2	0.8
S/random-composition/2G/R1/2/0.8	Stochastic	random-composition	2G	RPI-1	2	0.8
S/random-composition/512M/R1/2/1	Stochastic	random-composition	512M	RPI-1	2	1
S/random-composition/1G/R1/2/1	Stochastic	random-composition	1G	RPI-1	2	1
S/random-composition/2G/R1/2/1	Stochastic	random-composition	2G	RPI-1	2	1
S/random-composition/512M/R1/2/1.2	Stochastic	random-composition	512M	RPI-1	2	1.2
S/random-composition/1G/R1/2/1.2	Stochastic	random-composition	1G	RPI-1	2	1.2
S/random-composition/2G/R1/2/1.2	Stochastic	random-composition	2G	RPI-1	2	1.2
S/agenet-3/512M/L/4/0.8	Stochastic	agenet-3	512M	Laptop	4	0.8
S/agenet-3/1G/L/4/0.8	Stochastic	agenet-3	1G	Laptop	4	0.8
S/agenet-3/2G/L/4/0.8	Stochastic	agenet-3	2G	Laptop	4	0.8
S/agenet-3/512M/L/2/0.8	Stochastic	agenet-3	512M	Laptop	2	0.8
S/agenet-3/1G/L/2/0.8	Stochastic	agenet-3	1G	Laptop	2	0.8
S/agenet-3/2G/L/2/0.8	Stochastic	agenet-3	2G	Laptop	2	0.8
S/agenet-3/512M/L/6/0.8	Stochastic	agenet-3	512M	Laptop	6	0.8
S/agenet-3/1G/L/6/0.8	Stochastic	agenet-3	1G	Laptop	6	0.8
S/agenet-3/2G/L/6/0.8	Stochastic	agenet-3	2G	Laptop	6	0.8
S/agenet-3/512M/L/4/1	Stochastic	agenet-3	512M	Laptop	4	1
S/agenet-3/1G/L/4/1	Stochastic	agenet-3	1G	Laptop	4	1
S/agenet-3/2G/L/4/1	Stochastic	agenet-3	2G	Laptop	4	1
S/agenet-3/512M/L/2/1	Stochastic	agenet-3	512M	Laptop	2	1
S/agenet-3/1G/L/2/1	Stochastic	agenet-3	1G	Laptop	2	1
S/agenet-3/2G/L/2/1	Stochastic	agenet-3	2G	Laptop	2	1
S/agenet-3/512M/L/6/1	Stochastic	agenet-3	512M	Laptop	6	1
S/agenet-3/1G/L/6/1	Stochastic	agenet-3	1G	Laptop	6	1
S/agenet-3/2G/L/6/1	Stochastic	agenet-3	2G	Laptop	6	1
S/agenet-3/512M/L/4/1.2	Stochastic	agenet-3	512M	Laptop	4	1.2
S/agenet-3/1G/L/4/1.2	Stochastic	agenet-3	1G	Laptop	4	1.2
S/agenet-3/2G/L/4/1.2	Stochastic	agenet-3	2G	Laptop	4	1.2
S/agenet-3/512M/L/2/1.2	Stochastic	agenet-3	512M	Laptop	2	1.2
S/agenet-3/1G/L/2/1.2	Stochastic	agenet-3	1G	Laptop	2	1.2
S/agenet-3/2G/L/2/1.2	Stochastic	agenet-3	2G	Laptop	2	1.2
S/agenet-3/512M/L/6/1.2	Stochastic	agenet-3	512M	Laptop	6	1.2
S/agenet-3/1G/L/6/1.2	Stochastic	agenet-3	1G	Laptop	6	1.2
S/agenet-3/2G/L/6/1.2	Stochastic	agenet-3	2G	Laptop	6	1.2

D
Experiment results

D.1. B:Small-3

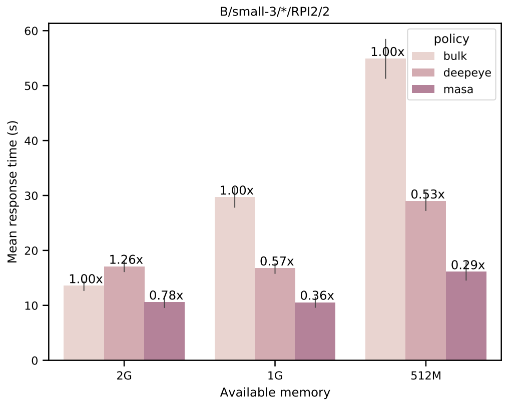


(a) Mean response time. Lower is better.

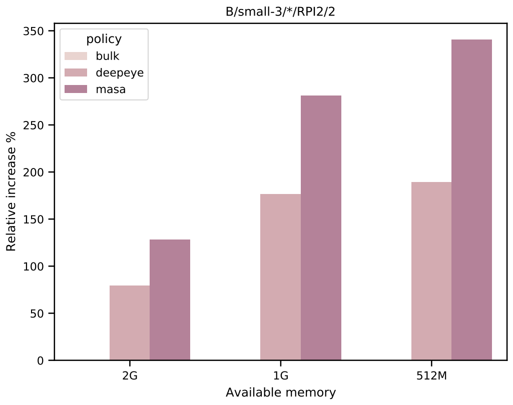


(b) Relative performance increase to `Bulk`. Higher is better.

Figure D.1: Response time of batch scenario I: comparisons of `Bulk`, `DeepEye` and MASA.

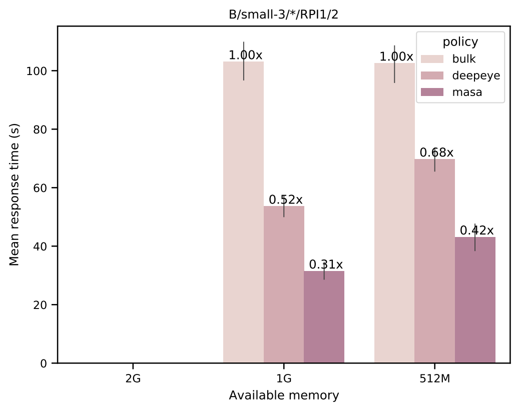


(a) Mean response time. Lower is better.

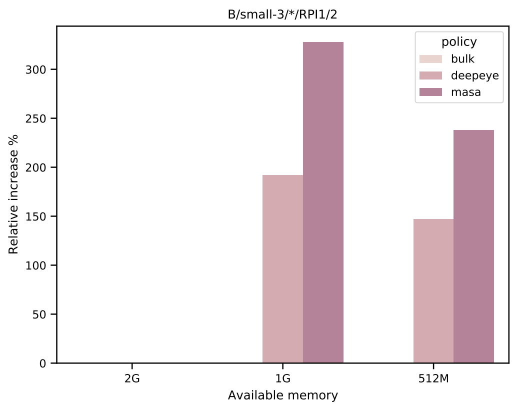


(b) Relative performance increase to `Bulk`. Higher is better.

Figure D.2: Response time of batch scenario I: comparisons of `Bulk`, `DeepEye` and MASA.



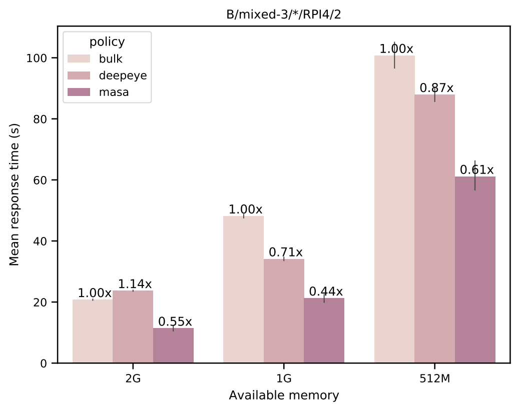
(a) Mean response time. Lower is better.



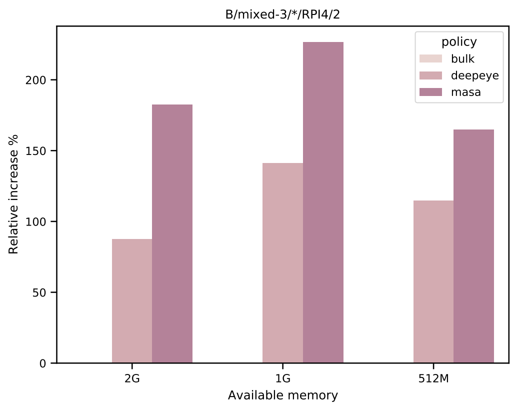
(b) Relative performance increase to `Bulk`. Higher is better.

Figure D.3: Response time of batch scenario I: comparisons of `Bulk`, `DeepEye` and MASA.

D.2. B:Mixed-3

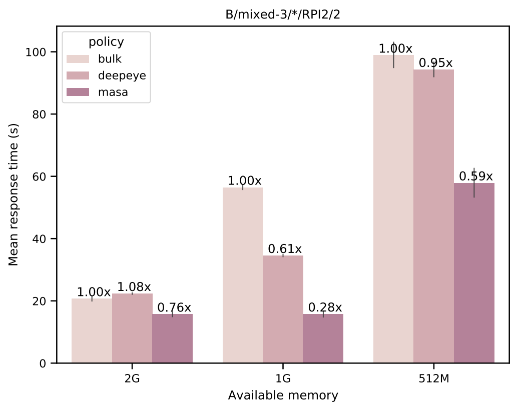


(a) Mean response time. Lower is better.

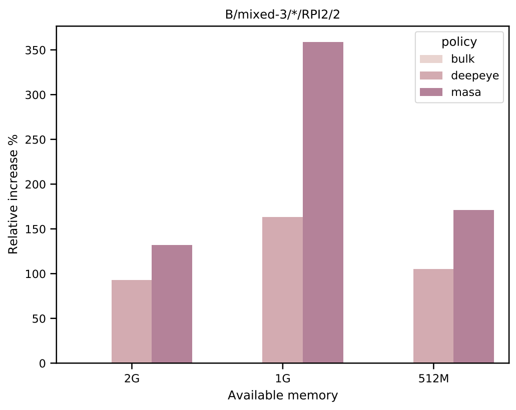


(b) Relative performance increase to `Bulk`. Higher is better.

Figure D.4: Response time of batch scenario II: comparisons of `Bulk`, `DeepEye` and MASA.

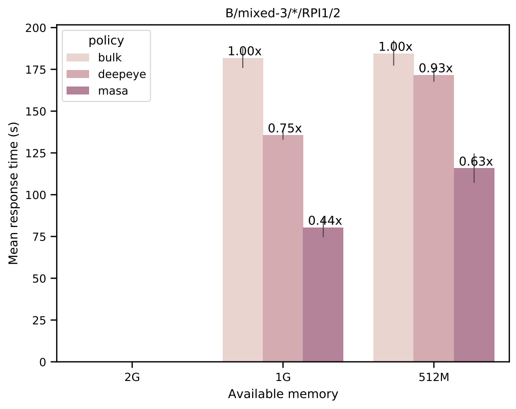


(a) Mean response time. Lower is better.

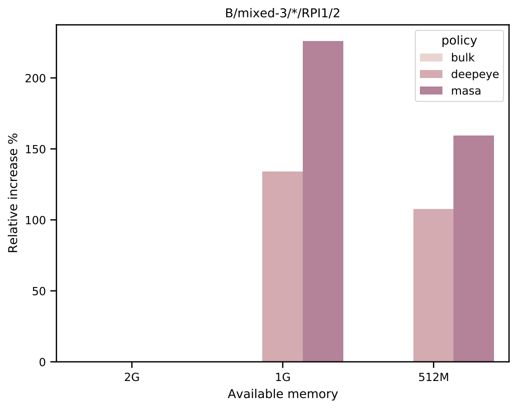


(b) Relative performance increase to `Bulk`. Higher is better.

Figure D.5: Response time of batch scenario II: comparisons of `Bulk`, `DeepEye` and MASA.



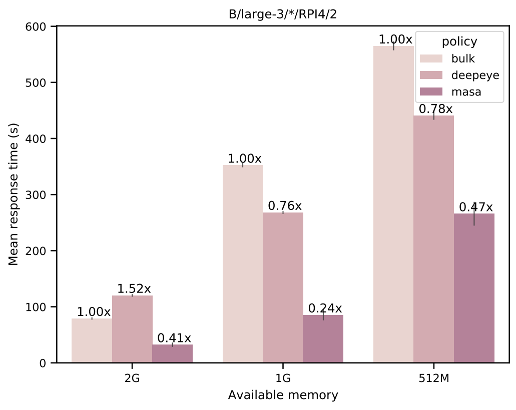
(a) Mean response time. Lower is better.



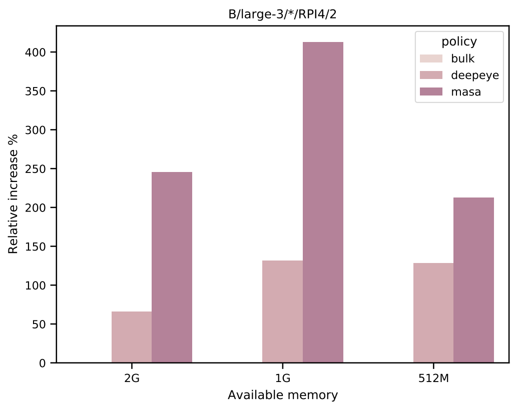
(b) Relative performance increase to `Bulk`. Higher is better.

Figure D.6: Response time of batch scenario II: comparisons of `Bulk`, `DeepEye` and MASA.

D.3. B:Large-3

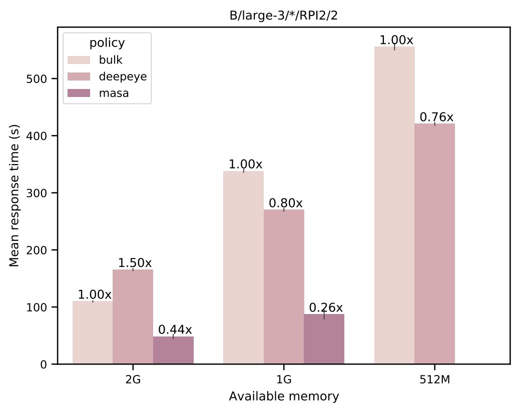


(a) Mean response time. Lower is better.

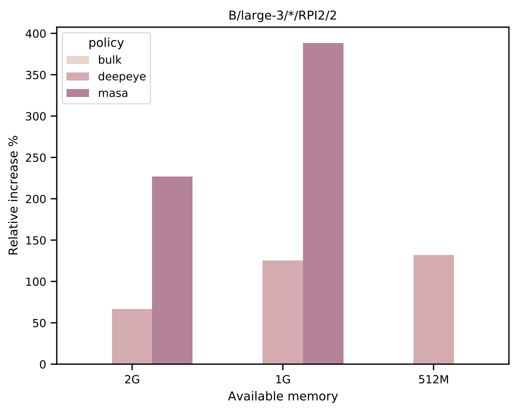


(b) Relative performance increase to `Bulk`. Higher is better.

Figure D.7: Response time of batch scenario III: comparisons of `Bulk`, `DeepEye` and MASA.

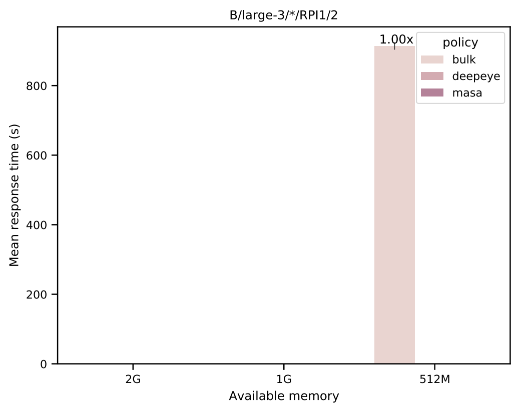


(a) Mean response time. Lower is better.

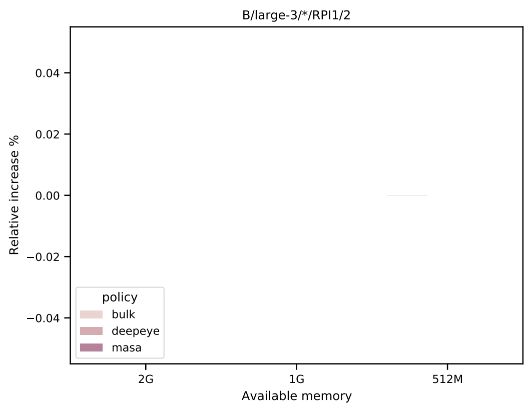


(b) Relative performance increase to `Bulk`. Higher is better.

Figure D.8: Response time of batch scenario III: comparisons of `Bulk`, `DeepEye` and MASA.



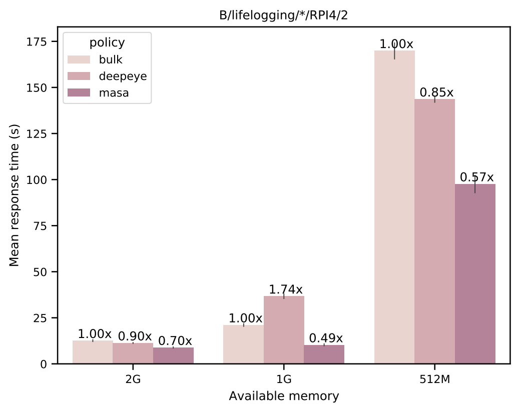
(a) Mean response time. Lower is better.



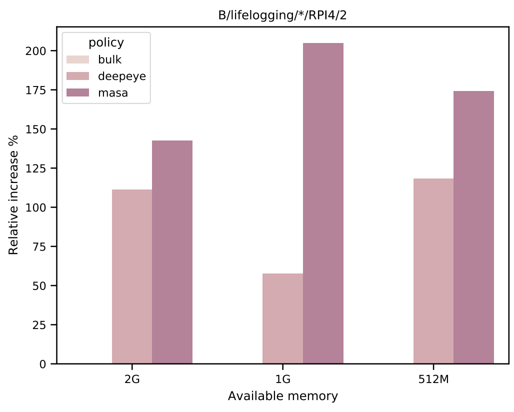
(b) Relative performance increase to `Bulk`. Higher is better.

Figure D.9: Response time of batch scenario III: comparisons of `Bulk`, `DeepEye` and MASA.

D.4. B:Lifelogging

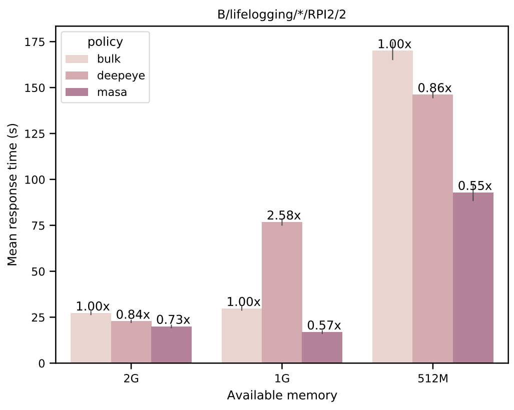


(a) Mean response time. Lower is better.

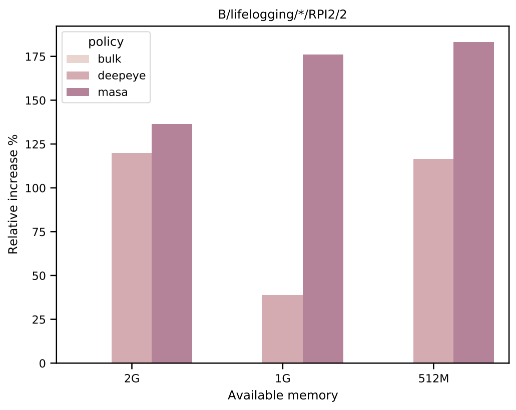


(b) Relative performance increase to `Bulk`. Higher is better.

Figure D.10: Response time of batch scenario IV: comparisons of `Bulk`, `DeepEye` and MASA.

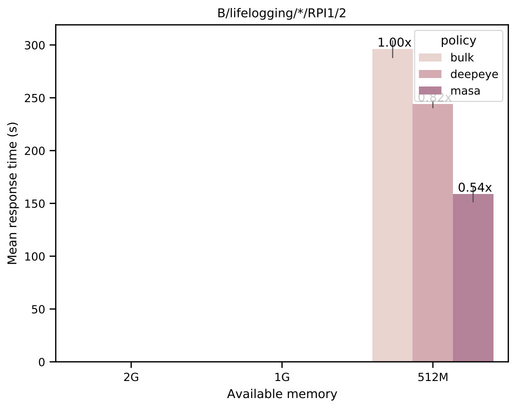


(a) Mean response time. Lower is better.

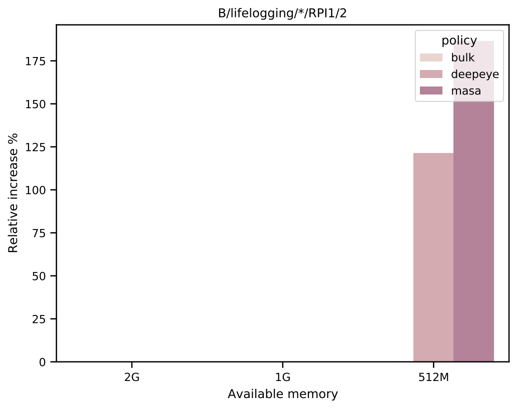


(b) Relative performance increase to `Bulk`. Higher is better.

Figure D.11: Response time of batch scenario IV: comparisons of `Bulk`, `DeepEye` and MASA.



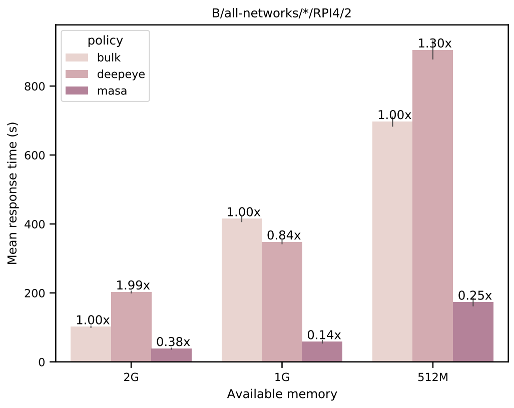
(a) Mean response time. Lower is better.



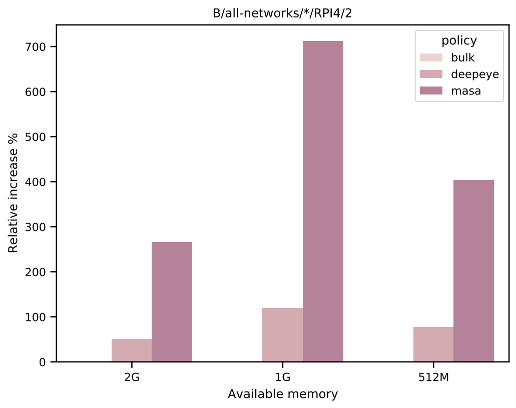
(b) Relative performance increase to `Bulk`. Higher is better.

Figure D.12: Response time of batch scenario IV: comparisons of `Bulk`, `DeepEye` and MASA.

D.5. B:All-networks

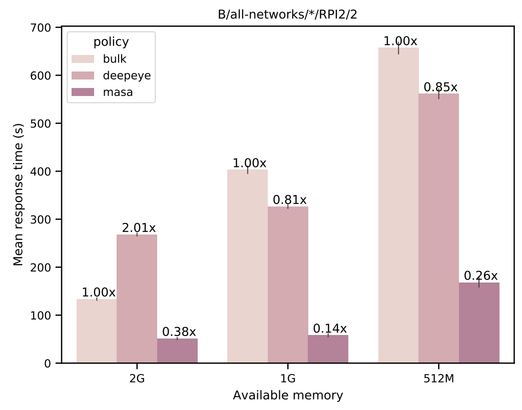


(a) Mean response time. Lower is better.

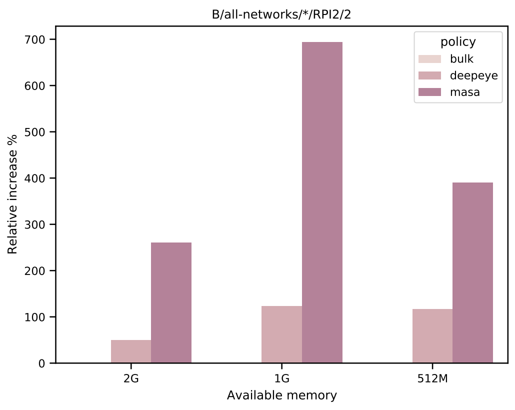


(b) Relative performance increase to `Bulk`. Higher is better.

Figure D.13: Response time of batch scenario V: comparisons of `Bulk`, `DeepEye` and MASA.

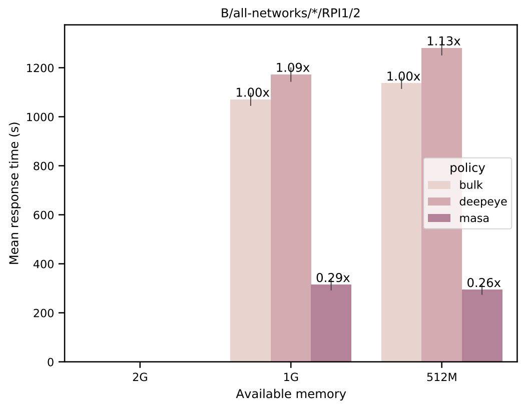


(a) Mean response time. Lower is better.

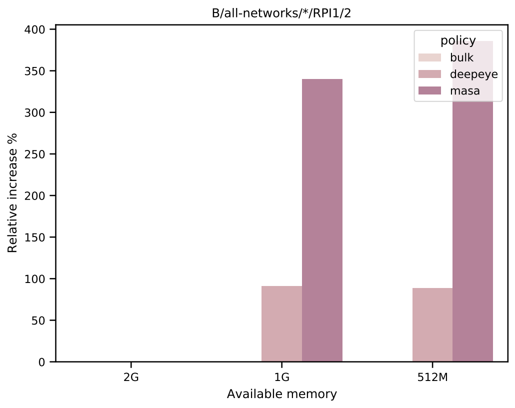


(b) Relative performance increase to `Bulk`. Higher is better.

Figure D.14: Response time of batch scenario V: comparisons of `Bulk`, `DeepEye` and MASA.



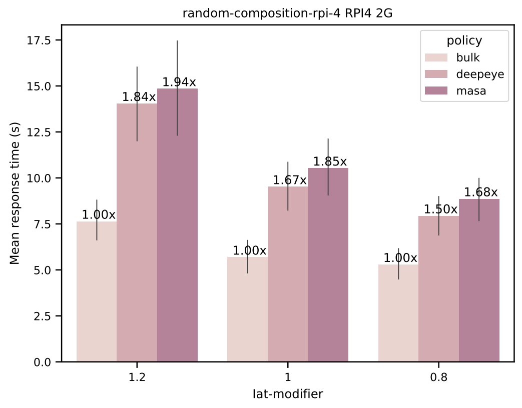
(a) Mean response time. Lower is better.



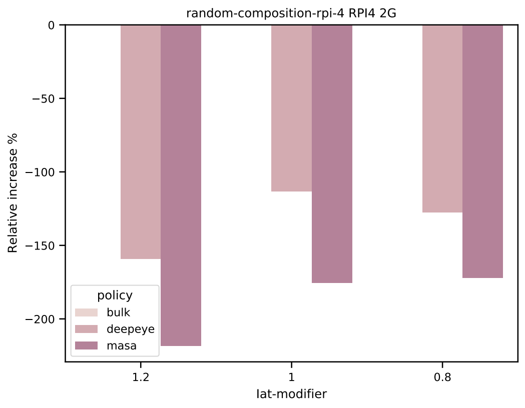
(b) Relative performance increase to `Bulk`. Higher is better.

Figure D.15: Response time of batch scenario V: comparisons of `Bulk`, `DeepEye` and MASA.

D.6. S:Random-composition

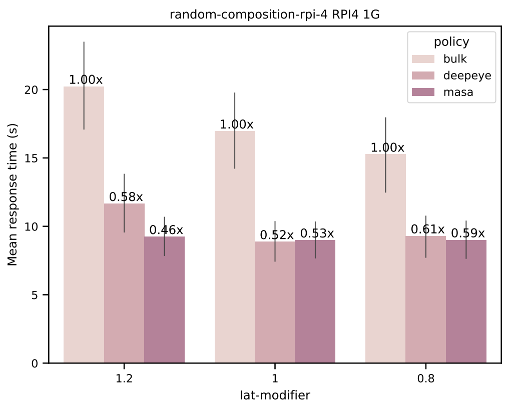


(a) Mean response time. Lower is better.

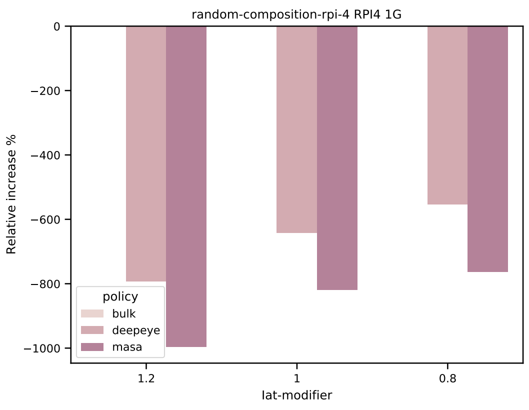


(b) Relative time to `Bulk`. Lower is better.

Figure D.16: Response time of stochastic scenario II: comparisons of `Bulk`, `DeepEye` and MASA.

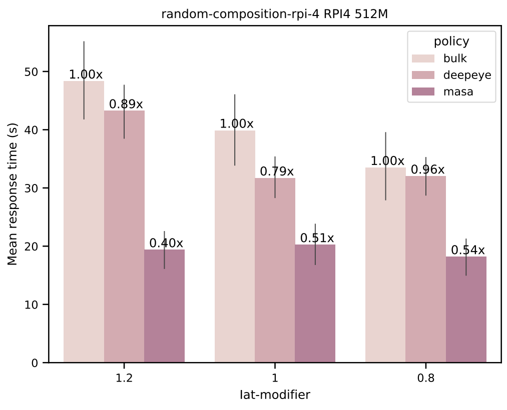


(a) Mean response time. Lower is better.

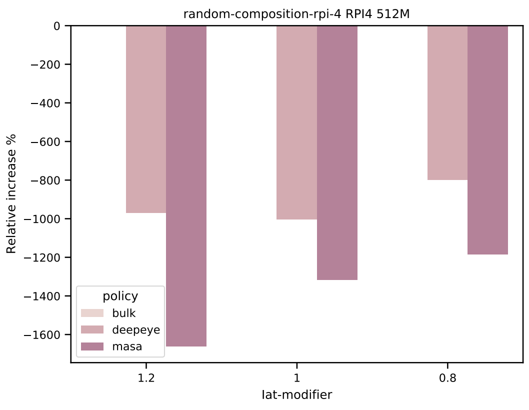


(b) Relative time to `Bulk`. Lower is better.

Figure D.17: Response time of stochastic scenario II: comparisons of `Bulk`, `DeepEye` and MASA.



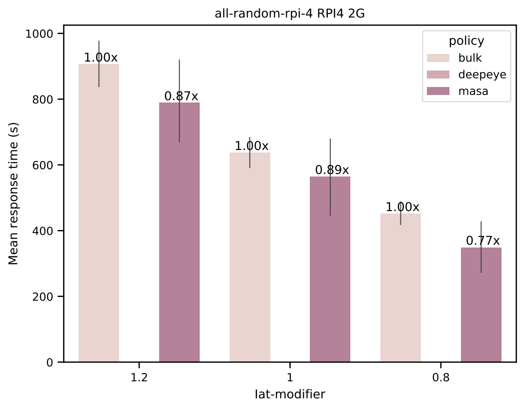
(a) Mean response time. Lower is better.



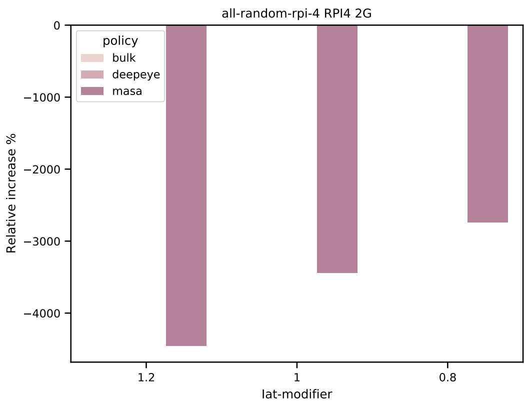
(b) Relative time to `Bulk`. Lower is better.

Figure D.18: Response time of stochastic scenario II: comparisons of `Bulk`, `DeepEye` and MASA.

D.7. S:All-Random

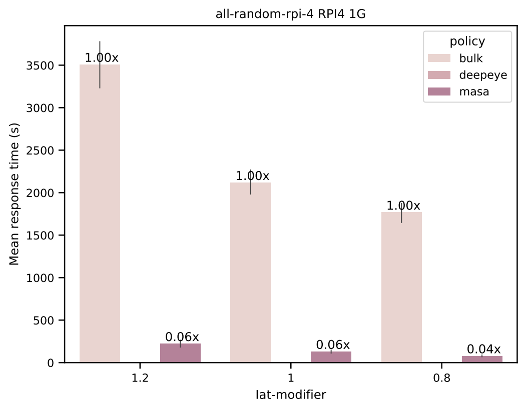


(a) Mean response time. Lower is better.

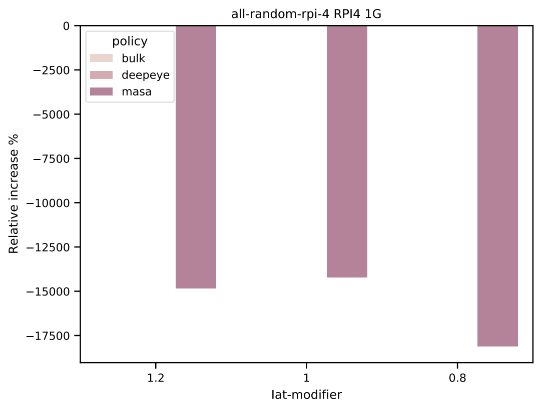


(b) Relative time to `Bulk`. Lower is better.

Figure D.19: Response time of stochastic scenario III: comparisons of `Bulk`, `DeepEye` and MASA.

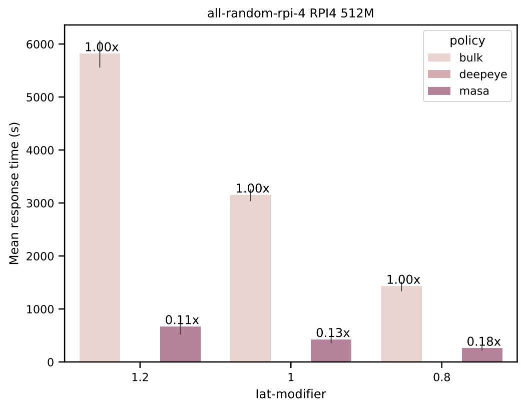


(a) Mean response time. Lower is better.

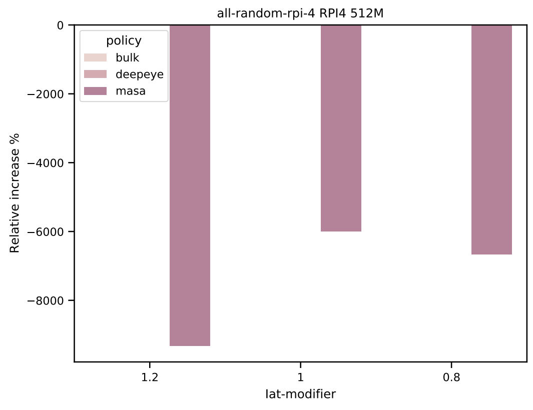


(b) Relative time to `Bulk`. Lower is better.

Figure D.20: Response time of stochastic scenario III: comparisons of `Bulk`, `DeepEye` and MASA.



(a) Mean response time. Lower is better.



(b) Relative time to `Bulk`. Lower is better.

Figure D.21: Response time of stochastic scenario III: comparisons of `Bulk`, `DeepEye` and MASA.

List of Figures

List of Tables

Bibliography

[1] Narrative clip 1 - order today. URL `http://getnarrative.com/narrative-clip-1`.

[2] Angeline Aguinaldo, Ping-yeh Chiang Alex, Gain Ameya, Patil Kolten, and Pearson Soheil. Compressing GANs using Knowledge Distillation.

[3] Amir H. Ashouri, Tarek S. Abdelrahman, and Alwyn Dos Remedios. Fast On-the-fly Retraining-free Sparsification of Convolutional Neural Networks. *Neurocomputing*, 370:56–69, Dec 2019. ISSN 0925-2312. doi: 10.1016/j.neucom.2019.08.063. URL `http://dx.doi.org/10.1016/j.neucom.2019.08.063`.

[4] Soroush Bateni and Cong Liu. Neuos: A latency-predictable multi-dimensional optimization framework for dnn-driven autonomous systems. In *USENIX ATC 20*, pages 371–385, 2020.

[5] Alfredo Canziani, Adam Paszke, and Eugenio Culurciello. An Analysis of Deep Neural Network Models for Practical Applications. pages 1–7, 2016. URL `http://arxiv.org/abs/1605.07678`.

[6] Tianqi Chen, Mu Li, Yutian Li, Min Lin, Naiyan Wang, Minjie Wang, Tianjun Xiao, Bing Xu, Chiyuan Zhang, and Zheng Zhang. Mxnet: A flexible and efficient machine learning library for heterogeneous distributed systems. *arXiv preprint arXiv:1512.01274*, 2015.

[7] Yu Cheng, Duo Wang, Pan Zhou, and Tao Zhang. A Survey of Model Compression and Acceleration for Deep Neural Networks. pages 1–10, 2019. URL `http://arxiv.org/abs/1710.09282`.

[8] Francois Chollet. Xception: Deep Learning with Depthwise Separable Convolutions. *SAE International Journal of Materials and Manufacturing*, 7(3):1251–1258, 2014. ISSN 19463979. doi: 10.4271/2014-01-0975.

[9] Stephen A Cook. The complexity of theorem-proving procedures. In *Proceedings of the third annual ACM symposium on Theory of computing*, pages 151–158, 1971.

[10] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. (Mlm), 2018. URL `http://arxiv.org/abs/1810.04805`.

[11] Mariella Dimiccoli, Marc Bolaños, Estefania Talavera, Maedeh Aghaei, Stavri G Nikolov, and Petia Radeva. Sr-clustering: Semantic regularized clustering for egocentric photo streams segmentation. *Computer Vision and Image Understanding*, 155:55–69, 2017.

[12] Z Eth and Dan Alistarh. M c d q. (2015):1–21, 2018.

[13] Biyi Fang, Xiao Zeng, and Mi Zhang. NestDNN: Resource-aware multi-tenant on-device deep learning for continuous mobile vision. *MOBICOM*, pages 115–127, 2018. doi: 10.1145/3241539.3241559.

[14] Sachin Sudhakar Farfade, Mohammad J Saberian, and Li-Jia Li. Multi-view face detection using deep convolutional neural networks. In *ACM ICMR*, pages 643–650, 2015.

[15] Ali Farhadi, Franziska Roesner, and Yejin Choi. e k a f. 2019.

[16] Google. Google Performance Tools. `https://github.com/gperftools/gperftools`, 2011.

[17] David Gries, Alain J Martin, Jan LA van de Snepscheut, and Jan Tijmen Udding. An algorithm for transitive reduction of an acyclic graph. *Science of Computer Programming*, 12(2):151–155, 1989.

[18] Seungyeop Han, Haichen Shen, Matthai Philipose, Sharad Agarwal, Alec Wolman, and Arvind Krishnamurthy. MCDNN: An approximation-based execution framework for deep stream processing under resource constraints. *MobiSys*, pages 123–136, 2016. doi: 10.1145/2906388.2906396.

[19] Yihui He, Ji Lin, Zhijian Liu, Hanrui Wang, Li-Jia Li, and Song Han. AMC: AutoML for Model Compression and Acceleration on Mobile Devices. *Eccv*, pages 815–832, 2018.

[20] Gao Huang. CondenseNet: An Efficient DenseNet using Learned Group Convolutions. *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 15:2752–2761, 2018. ISSN 01479601.

[21] Yanping Huang, Youlong Cheng, Ankur Bapna, Orhan Firat, Mia Xu Chen, Dehao Chen, HyoukJoong Lee, Jiquan Ngiam, Quoc V. Le, Yonghui Wu, and Zhifeng Chen. GPipe: Efficient Training of Giant Neural Networks using Pipeline Parallelism. pages 103–112, 2019.

[22] Loc N Huynh, Youngki Lee, and Rajesh Krishna Balan. Deepmon: Mobile gpu-based deep learning framework for continuous vision applications. In *MobiSys*, pages 82–95, 2017.

[23] Yangqing Jia, Evan Shelhamer, Jeff Donahue, Sergey Karayev, Jonathan Long, Ross Girshick, Sergio Guadarrama, and Trevor Darrell. Caffe: Convolutional architecture for fast feature embedding. In *ACM International Conference on Multimedia*, pages 675–678, 2014.

[24] Ziheng Jiang, Tianqi Chen, and Mu Li. Efficient deep learning inference on edge devices. In *ACM SysML*, 2018.

[25] Vidur Joshi, Matthew Peters, and Mark Hopkins. Extending a parser to distant domains using a few dozen partially annotated examples. *ACL 2018 - 56th Annual Meeting of the Association for Computational Linguistics, Proceedings of the Conference (Long Papers)*, 1:1190–1199, 2018. doi: 10.18653/v1/p18-1110.

[26] Arthur B Kahn. Topological sorting of large networks. *Communications of the ACM*, 5(11):558–562, 1962.

[27] Aditya Khosla, Akhil S Raju, Antonio Torralba, and Aude Oliva. Understanding and predicting image memorability at a large scale. In *IEEE ICCV*, pages 2390–2398, 2015.

[28] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. Imagenet classification with deep convolutional neural networks. In *NIPS*, pages 1106–1114, 2012. URL `http://papers.nips.cc/paper/4824-imagenet-classification-with-deep-convolutional-neural-networks`.

[29] Philippe Laborie, Jérôme Rogerie, Paul Shaw, and Petr Vilím. Ibm ilog cp optimizer for scheduling. *Constraints*, 23(2):210–250, 2018.

[30] Guillaume Lample and Alexis Conneau. Cross-lingual Language Model Pretraining. 2019. URL `http://arxiv.org/abs/1901.07291`.

[31] Nicholas D. Lane, Sourav Bhattacharya, Akhil Mathur, Petko Georgiev, Claudio Forlivesi, and Fahim Kawsar. Squeezing Deep Learning into Mobile and Embedded Devices. *IEEE Pervasive Computing*, 16(3):82–88, 2017. ISSN 15361268. doi: 10.1109/MPRV.2017.2940968.

[32] Yann LeCun, Bernhard E. Boser, John S. Denker, Donnie Henderson, Richard E. Howard, Wayne E. Hubbard, and Lawrence D. Jackel. Backpropagation applied to handwritten zip code recognition. *Neural Computation*, 1(4):541–551, 1989. doi: 10.1162/neco.1989.1.4.541.

[33] Matthew LeMay, Shijian Li, and Tian Guo. Perseus: Characterizing performance and cost of multi-tenant serving for cnn models. In *IEEE IC2E*, pages 66–72, 2020.

[34] Gil Levi and Tal Hassncer. Age and gender classification using convolutional neural networks. In *IEEE CVPRW*, page 34–42, 6 2015. ISBN 978-1-4673-6759-2. doi: 10.1109/CVPRW.2015.7301352. URL `http://ieeexplore.ieee.org/document/7301352/`.

[35] Gil Levi and Tal Hassner. Emotion recognition in the wild via convolutional neural networks and mapped binary patterns. In *ACM ICMI*, pages 503–510, 2015.

[36] En Li, Liekang Zeng, Zhi Zhou, and Xu Chen. Edge AI: On-Demand Accelerating Deep Neural Network Inference via Edge Computing. *IEEE Transactions on Wireless Communications*, 19(1):1–1, 2019. ISSN 1536-1276. doi: 10.1109/twc.2019.2946140.

[37] Xiaodong Liu, Pengcheng He, Weizhu Chen, and Jianfeng Gao. Multi-Task Deep Neural Networks for Natural Language Understanding. pages 4487–4496, 2019. doi: 10.18653/v1/p19-1441.

[38] Yinhan Liu, Myle Ott, Naman Goyal, Jingfei Du, Mandar Joshi, Danqi Chen, Omer Levy, Mike Lewis, Luke Zettlemoyer, and Veselin Stoyanov. RoBERTa: A Robustly Optimized BERT Pretraining Approach. (1), 2019. URL `http://arxiv.org/abs/1907.11692`.

[39] Xiaolin Lu, Il Han Kim, Ariton Xhafa, Jianwei Zhou, and Kaichien Tsai. Reaching 10-years of battery life for industrial IoT wireless sensor networks. *IEEE Symposium on VLSI Circuits, Digest of Technical Papers*, 1:C66–C67, 2017. doi: 10.23919/VLSIC.2017.8008550.

[40] Mohammad Saeid Mahdavinejad, Mohammadreza Rezvan, Mohammadamin Barekatain, Peyman Adibi, Payam Barnaghi, and Amit P. Sheth. Machine learning for internet of things data analysis: a survey, 2018. ISSN 23528648.

[41] Vicent Sanz Marco, Ben Taylor, Zheng Wang, and Yehia Elkhatib. Optimizing deep learning inference on embedded systems through adaptive model selection. *ACM TECS*, 19(1):1–28, 2020.

[42] Akhil Mathurz, Nicholas D. Lanezy, Sourav Bhattacharyaz, Aidan Boranz, Claudio Forlivesiz, and Fahim Kawsarz. DeepEye: Resource efficient local execution of multiple deep vision models using wearable commodity hardware. *MobiSys*, pages 68–81, 2017. doi: 10.1145/3081333.3081359.

[43] Kais Mekki, Eddy Bajic, Frederic Chaxel, and Fernand Meyer. A comparative study of LPWAN technologies for large-scale IoT deployment. *ICT Express*, 2019. ISSN 24059595. doi: 10.1016/j.icte.2017.12.005.

[44] Decebal Constantin Mocanu, Elena Mocanu, Peter Stone, Phuong H. Nguyen, Madeleine Gibescu, and Antonio Liotta. Scalable training of artificial neural networks with adaptive sparse connectivity inspired by network science. *Nature Communications*, 9(1), 2018. ISSN 20411723. doi: 10.1038/s41467-018-04316-3.

[45] Mehdi Mohammadi, Ala Al-Fuqaha, Sameh Sorour, and Mohsen Guizani. Deep learning for IoT big data and streaming analytics: A survey. *IEEE Communications Surveys and Tutorials*, 20(4):2923–2960, 2018. ISSN 1553877X. doi: 10.1109/COMST.2018.2844341.

[46] Nicholas Nethercote and Julian Seward. Valgrind: a framework for heavyweight dynamic binary instrumentation. In Jeanne Ferrante and Kathryn S. McKinley, editors, *ACM SIGPLAN*, pages 89–100, 2007.

[47] Wei Niu, Xiaolong Ma, Sheng Lin, Shihao Wang, Xuehai Qian, Xue Lin, Yanzhi Wang, and Bin Ren. Patdnn: Achieving real-time dnn execution on mobile devices with pattern-based weight pruning. In *ASPLOS*, pages 907–922, 2020.

[48] S S Ogden and T Guo. MODI: Mobile Deep Inference Made Efficient by Edge Computing. *{USENIX} Workshop on Hot Topics in Edge Computing (HotEdge 18)*, (1):7, 2018. URL `http://tianguo.info/publications.html`.

[49] Samuel S. Ogden and Tian Guo. Characterizing the Deep Neural Networks Inference Performance of Mobile Applications. *USENIX, Workshop on Hot Topics in Edge Computing (HotEdge 18)*, 2019.

[50] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, et al. Pytorch: An imperative style, high-performance deep learning library. In *NIPS*, pages 8026–8037, 2019.

[51] Paul Purdom. A transitive closure algorithm. *BIT Numerical Mathematics*, 10(1):76–94, 1970.

[52] Alec Radford and Tim Salimans. Improving Language Understanding by Generative Pre-Training. *OpenAI*, pages 1–12, 2018.

[53] Alec Radford, Jeffrey Wu, Rewon Child, David Luan, Dario Amodei, and Ilya Sutskever. Language Models are Unsupervised Multitask Learners. 2018.

[54] Samyam Rajbhandari, Jeff Rasley, Olatunji Ruwase, and Yuxiong He. ZeRO: Memory Optimization Towards Training A Trillion Parameter Models. pages 1–17, 2019. URL `http://arxiv.org/abs/1910.02054`.

[55] Joseph Redmon and Ali Farhadi. Yolo9000: better, faster, stronger. In *CVPR*, pages 7263–7271, 2017.

[56] Mark Sandler, Andrew Howard, Menglong Zhu, Andrey Zhmoginov, and Liang Chieh Chen. MobileNetV2: Inverted Residuals and Linear Bottlenecks. *Proceedings of the IEEE Computer Society Conference on Computer Vision and Pattern Recognition*, pages 4510–4520, 2018. ISSN 10636919. doi: 10.1109/CVPR.2018.00474.

[57] Victor Sanh, Lysandre Debut, Julien Chaumond, and Thomas Wolf. DistilBERT, a distilled version of BERT: smaller, faster, cheaper and lighter. pages 2–6, 2019. URL `http://arxiv.org/abs/1910.01108`.

[58] Mohammad Shoeybi, Mostofa Patwary, Raul Puri, Patrick LeGresley, Jared Casper, and Bryan Catanzaro. Megatron-LM: Training Multi-Billion Parameter Language Models Using Model Parallelism. 2019. URL `http://arxiv.org/abs/1909.08053`.

[59] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. In Yoshua Bengio and Yann LeCun, editors, *ICLR*, 2015.

[60] Vishwanath A. Sindagi and Vishal M. Patel. A survey of recent advances in cnn-based single image crowd counting and density estimation. *Pattern Recognit. Lett.*, 107:3–16, 2018.

[61] Rashmi Sharan Sinha, Yiqiao Wei, and Seung Hoon Hwang. A survey on LPWA technology: LoRa and NB-IoT, 2017. ISSN 24059595.

[62] Estefania Talavera, Mariella Dimiccoli, Marc Bolanos, Maedeh Aghaei, and Petia Radeva. R-clustering for egocentric video segmentation. In *Iberian Conference on Pattern Recognition and Image Analysis*, pages 327–336. Springer, 2015.

[63] Ben Taylor, Vicent Sanz Marco, Willy Wolff, Yehia Elkhatib, and Zheng Wang. Adaptive deep learning model selection on embedded systems. *Proceedings of the ACM SIGPLAN Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES)*, pages 31–43, 2018. doi: 10.1145/3211332.3211336.

[64] Surat Teerapittayanon, Bradley McDanel, and H. T. Kung. BranchyNet: Fast inference via early exiting from deep neural networks. *Proceedings - International Conference on Pattern Recognition*, 0:2464–2469, 2016. ISSN 10514651. doi: 10.1109/ICPR.2016.7900006.

[65] M.V. Valueva, N.N. Nagornov, P.A. Lyakhov, G.V. Valuev, and N.I. Chervyakov. Application of the residue number system to reduce hardware costs of the convolutional neural network implementation. *Mathematics and Computers in Simulation*, 177:232 – 243, 2020. ISSN 0378-4754. doi: https://doi.org/10.1016/j.matcom.2020.04.031. URL `http://www.sciencedirect.com/science/article/pii/S0378475420301580`.

[66] Carole Jean Wu, David Brooks, Kevin Chen, Douglas Chen, Sy Choudhury, Marat Dukhan, Kim Hazelwood, Eldad Isaac, Yangqing Jia, Bill Jia, Tommer Leyvand, Hao Lu, Yang Lu, Lin Qiao, Brandon Reagen, Joe Spisak, Fei Sun, Andrew Tulloch, Peter Vajda, Xiaodong Wang, Yanghan Wang, Bram Wasti, Yiming Wu, Ran Xian, Sungjoo Yoo, and Peizhao Zhang. Machine learning at facebook: Understanding inference at the edge. *Proceedings - 25th IEEE International Symposium on High Performance Computer Architecture, HPCA 2019*, pages 331–344, 2019. doi: 10.1109/HPCA.2019.00048.

[67] Carole-Jean Wu, David Brooks, Kevin Chen, Douglas Chen, Sy Choudhury, Marat Dukhan, Kim Hazelwood, Eldad Isaac, Yangqing Jia, Bill Jia, et al. Machine learning at facebook: Understanding inference at the edge. In *IEEE HPCA*, pages 331–344, 2019.

[68] Y. Xiang and H. Kim. Pipelined data-parallel cpu/gpu scheduling for multi-dnn real-time inference. In *IEEE RTSS*, pages 392–405, 2019.

[69] Mengwei Xu, Mengze Zhu, Yunxin Liu, Felix Xiaozhu Lin, and Xuanzhe Liu. Deepcache: Principled cache for mobile deep vision. In *MobiCom*, pages 129–144, 2018.

[70] Xiaowei Xu, Yukun Ding, Sharon Xiaobo Hu, Michael Niemier, Jason Cong, Yu Hu, and Yiyu Shi. Scaling for edge inference of deep neural networks/639/166/987/639/705/117 perspective. *Nature Electronics*, 1(4):216–222, 2018. ISSN 25201131. doi: 10.1038/s41928-018-0059-3.

[71] Wenzhu Yang, Lilei Jin, Sile Wang, Zhenchao Cu, Xiangyang Chen, and Liping Chen. Thinning of convolutional neural network with mixed pruning. *IET Image Processing*, 13(5):779–784, 2019. ISSN 17519659. doi: 10.1049/iet-ipr.2018.6191.

[72] Zhilin Yang, Zihang Dai, Yiming Yang, Jaime Carbonell, Ruslan Salakhutdinov, and Quoc V. Le. XLNet: Generalized Autoregressive Pretraining for Language Understanding. (NeurIPS):1–11, 2019.

[73] Jianming Zhang, Shugao Ma, Mehrnoosh Sameki, Stan Sclaroff, Margrit Betke, Zhe Lin, Xiaohui Shen, Brian Price, and Radomir Mech. Salient object subitizing. In *IEEE CVPR*, pages 4045–4054, 2015.

[74] Wei Zhang, Kan Liu, Weidong Zhang, Youmei Zhang, and Jason Gu. Deep Neural Networks for wireless localization in indoor and outdoor environments. *Neurocomputing*, 194:279–287, 2016. ISSN 18728286. doi: 10.1016/j.neucom.2016.02.055. URL `http://dx.doi.org/10.1016/j.neucom.2016.02.055`.

[75] X. Zhang, J. Zou, K. He, and J. Sun. Accelerating very deep convolutional networks for classification and detection. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 38(10):1943–1955, 2016.

[76] Xiangyu Zhang. ShuffleNet: An Extremely Efficient Convolutional Neural Network for Mobile Devices. *ShuffleNet: An Extremely Efficient Convolutional Neural Network for Mobile Devices*, pages 6848–6856, 2017. doi: 10.4324/9780203491348.

[77] Bolei Zhou, Agata Lapedriza, Jianxiong Xiao, Antonio Torralba, and Aude Oliva. Learning deep features for scene recognition using places database. In *NIPS*, pages 487–495, 2014.