Contents lists available at ScienceDirect

Future Generation Computer Systems

DNNShifter: An efficient DNN pruning system for edge computing

Bailey J. Eccles [a],*, Philip Rodgers [b], Peter Kilpatrick [c], Ivor Spence [c], Blesson Varghese [a]

[a] School of Computer Science, University of St Andrews, Jack Cole Building, St Andrews, KY16 9SX, Fife, Scotland, United Kingdom
[b] Autonomous Networking Research & Innovation Department, Rakuten Mobile, Rakuten Crimson House, 1-14-1 Tamagawa, Setagaya-ku, Tokyo, 158-0094, Tokyo, Japan
[c] School of Electronics, Electrical Engineering & Computer Science, Queen's University Belfast, Computer Science Building, 16A Malone Road, Belfast, BT9 5BN, Co. Antrim, Northern Ireland, United Kingdom

ARTICLE INFO

ABSTRACT

Deep neural networks (DNNs) underpin many machine learning applications. Production quality DNN models achieve high inference accuracy by training millions of DNN parameters which has a significant resource footprint. This presents a challenge for resources operating at the extreme edge of the network, such as mobile and embedded devices that have limited computational and memory resources. To address this, models are pruned to create lightweight, more suitable variants for these devices. Existing pruning methods are unable to provide similar quality models compared to their unpruned counterparts without significant time costs and overheads or are limited to offline use cases. Our work rapidly derives suitable model variants while maintaining the accuracy of the original model. The model variants can be swapped quickly when system and network conditions change to match workload demand. This paper presents DNNShifter, an end-to-end DNN training, spatial pruning, and model switching system that addresses the challenges mentioned above. At the heart of DNNShifter is a novel methodology that prunes sparse models using structured pruning - combining the accuracy-preserving benefits of unstructured pruning with runtime performance improvements of structured pruning. The pruned model variants generated by DNNShifter are smaller in size and thus faster than dense and sparse model predecessors, making them suitable for inference at the edge while retaining near similar accuracy as of the original dense model. DNNShifter generates a portfolio of model variants that can be swiftly interchanged depending on operational conditions. DNNShifter produces pruned model variants up to 93x faster than conventional training methods. Compared to sparse models, the pruned model variants are up to 5.14x smaller and have a 1.67x inference latency speedup, with no compromise to sparse model accuracy. In addition, DNNShifter has up to 11.9x lower overhead for switching models and up to 3.8x lower memory utilisation than existing approaches. DNNShifter is available for public use from https://github.com/blessonvar/DNNShifter.

1. Introduction

Deep neural networks (DNNs) are machine learning (ML) models comprising a sequence of layers, such as convolution and linear. Such models find application in object detection and image classification due to their high accuracy [1]. Production quality DNN models trained on standard datasets contain a large number of parameters. For example, VGG-16 [2] trained on the ImageNet [3] dataset contains 138M parameters. Such models have significant CPU or memory resource requirements and, consequently, high energy consumption for training and inference. Hence, they are suited for resource-rich environments like cloud or high-performance computing sites. These DNNs can-

not be adopted for relatively resource-constrained environments, such as the (extreme) network edge dominated by mobile and embedded devices [4].

Edge environments cannot support production quality DNNs due to compute [4], memory [5] and energy [6] constraints. Therefore, approaches for deriving lightweight DNN model variants from production quality DNNs using (i) neural architecture search (NAS) [7] and (ii) pre-trained model compression [5] have been proposed. These approaches have a two-fold *limitation*. Firstly, they are time-consuming and costly [7]. For example, the NasNet [8] search requires four days of computation on 500 data centre-grade GPUs to find optimal model variants.

---

* Corresponding author.
E-mail addresses: bje1@st-andrews.ac.uk (B.J. Eccles), philip.rodgers@rakuten.com (P. Rodgers), p.kilpatrick@qub.ac.uk (P. Kilpatrick), i.spence@qub.ac.uk (I. Spence), blesson@st-andrews.ac.uk (B. Varghese).

Secondly, the model variants obtained from these approaches are static. The models are optimised against specific objectives, such as accuracy, inference latency, or model size [7]. Therefore, they cannot be used on the edge to meet the requirements of varying operational conditions, such as changing resource utilisation levels [9,10].

Existing NAS and compression approaches cannot be used for rapidly producing models, and the models produced by these approaches cannot be adapted to suit changing operational conditions on the edge. The research reported in this paper is therefore focused towards addressing the above limitations and surmounts the following challenges:

Challenge 1 - Rapidly generating a range of DNNs suited for different operational conditions and heterogeneous edge resources: ML applications that run on the edge will need to execute pre-trained DNNs. Training a DNN tailored to the edge resource using approaches, such as NAS, is not suitable as they are time and energy-consuming [7]. Alternatively, compressing a pre-trained DNN using knowledge distillation [11] is based on trial and error, or quantisation [12] that requires specialised hardware or libraries that may not be available in edge environments.

Challenge 2 - Spatial compression of DNN models while maintaining accuracy: DNN compression methods, such as structured pruning [13] or re-parameterisation [14], can significantly reduce the size of a model. However, these methods remove parameters essential to model accuracy. For example, convolutional layers are sensitive to pruning, and even small degrees of pruning can negatively impact accuracy [13]. Consequently, the compressed model is fine-tuned after pruning using computationally expensive methods to regain accuracy, which can take up to 3 times the original training time [5].

Challenge 3 - On-demand switching of compressed DNNs to adapt to dynamic operational environments: DNNs used at the edge will need to seamlessly adapt to changing conditions in real time by switching models on-demand to match model inference performance thresholds. However, existing approaches will incur a downtime in the order of minutes [7] to hours [15] for identifying and deploying a suitable model that meets the desired performance [10].

This paper presents `DNNShifter`, a framework that utilises production quality sparse models to generate a portfolio of spatially compressed model variants with high accuracy in real time. This is achieved by proposing a novel method that uses structured pruning of highly sparse models. This results in pruned models with a smaller resource footprint and the same model accuracy as the original sparse model. This method fundamentally differs from the commonly reported structured pruning methods that prune pre-trained dense models and negatively impact accuracy. The portfolio of models that are generated by our method can be used to adapt to match a range of operational runtime requirements. This is achieved by low overhead switching from one model to another in the portfolio at runtime. `DNNShifter` makes the following three research **contributions**:

(1) A time and resource-efficient guided DNN model-training, pruning, and runtime switching pipeline that creates a portfolio of spatially pruned model variants comparable to a typical DNN model training routine using NAS. `DNNShifter` generates a portfolio of pruned model variants up to 93x faster than state-of-the-art methods.

(2) A novel pruning method to compress highly sparse DNN models, resulting in accurate and spatially compact pruned model variants suited for edge resources with low inference latency. `DNNShifter` pruned model variants are up to 5.14x smaller and have up to 1.67x and 1.45x faster CPU and GPU inference latencies, respectively. In addition, the pruned model variants can be obtained orders of magnitude faster when compared to existing structured pruning methods and have higher accuracy for a given model size.

(3) A low-overhead method that switches from one model variant to another on-demand at the edge to match a range of operational runtime requirements. `DNNShifter` has up to 11.9x lower overhead for switching model variants with up to 3.8x lower memory utilisation than existing approaches.

The remainder of this paper is organised as follows. Section 2 discusses related work. Section 3 presents the `DNNShifter` framework. Section 4 presents experimental results. Section 5 concludes the paper by discussing system limitations.
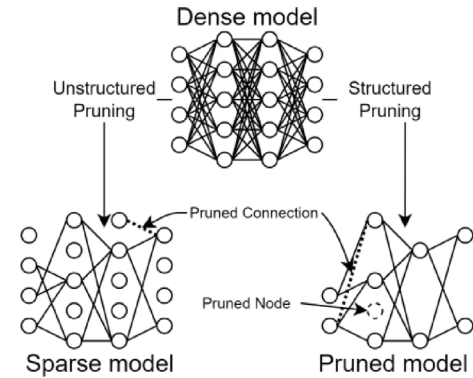

Fig. 1. Obtaining sparse and pruned models from pruning a dense model.

2. Related work

Approaches for DNN compression aim to improve the resource efficiency of models by reducing their computational and memory utilisations while preserving the model's accuracy. Techniques such as model pruning, quantisation, and knowledge distillation leverage different properties of a DNN that inherently lend towards compression. As a result, a compressed model that is either smaller, faster or both compared to the original model, is produced. These approaches typically produce a single compressed model.

Techniques such as NAS, on the other hand, generate a portfolio of compressed models from a search space that suits the requirements of constrained resources [8,16–18]. However, NAS is computationally expensive because it trains and evaluates many candidate models (up to thousands) before identifying the optimal set of compressed models. *Our work is positioned at the intersection of DNN compression and NAS, where a more time and resource-efficient compression pipeline than NAS is developed to generate a portfolio of highly compressed models, which serves a range of resource requirements as seen in edge environments.* This section considers the key contributions of each DNN compression method by providing an overview of their strengths and weaknesses and comparing their features. More recent work is considered to highlight the novelty of the `DNNShifter` framework we propose in addition to presenting baseline methods to compare `DNNShifter` in the experiments presented in Section 4.

2.1. Unstructured pruning and sparse models

Unstructured pruning masks select individual parameters of the DNN by setting their weight to zero [19–22]. Existing methods such as Lottery Ticket Hypothesis (LTH) [23] demonstrate that introducing sparsity via unstructured pruning early into training can lead to similar or higher final accuracy as a dense model. In addition, with suitable hardware and sparse matrix libraries [24], sparse models can accelerate model training, thus, reducing time and energy costs [25]. The concept of LTH has motivated a large collection of unstructured pruning methods [20,22,26,27]. However, resource-constrained environments usually do not support libraries to leverage any performance benefits of sparse models [28]. Furthermore, the zeroed weights within the sparse model do not reduce the memory footprint but create irregular memory accesses that will degrade inference performance on conventional CPUs. Unstructured pruning research typically focuses on improving sparse model accuracy rather than considering compute performance considerations [26]. *Sparse models are the starting point for our work.* The `DNNShifter` framework removes sparse data structures within these models in a disciplined manner. In other words, we spatially remove the zeroed weights, thereby reducing the model size and inference latency while maintaining the same accuracy as the original sparse model.

2.2. Structured pruning and re-parameterisation

As shown in Fig. 1, structured pruning spatially removes groups of parameters, such as convolutional filters [13,15,29–31]. A ranking algorithm is employed to identify filters that contribute the least to accuracy loss if removed. Structured pruning removes these filters, resulting in models with lower memory, energy, and inference footprint. However, structured pruning is time-consuming because: (i) the filters that can be removed need to be identified given the thousands of pruning combinations [15,29], and (ii) the parameters that remain after pruning are fine-tuned to recover the accuracy lost while pruning [5, 20]. Thus, on-demand compression cannot be achieved using structured pruning.

However, `DNNShifter` achieves structured pruning in (near) real-time by leveraging the following observations of sparse models: (i) zeroed weights are inherently ranked as prunable, and (ii) pruning of zeroed weights does not reduce model accuracy. Furthermore, structured re-parameterisation can be combined with structured pruning to further optimise a model by modifying the underlying architecture for a target device. For example, RepVGG [14] restructures ResNets into a VGG-like architecture and improves GPU utilisation.

2.3. Dynamic DNNs

Dynamic DNNs improve inference efficiency by adapting a DNN for different operational conditions [32]. Methods such as skipping layers [33] and connections [34] or early-exiting [32] decrease inference latency at the cost of inference accuracy. Although dynamic DNNs offer the advantage of using any sub-model from within a single model, there are no spatial benefits since the entire model runs in memory, even for a small sub-model [32]. Alternatively, `DNNShifter` provides both inference and spatial benefits and leverages in-memory compression of multiple sparse models to facilitate on-demand switching of models to suit runtime requirements.

2.4. Other compression methods

Other compression methods, namely quantisation and knowledge distillation, are presented in the literature for DNN compression. Quantisation reduces the bit precision of parameters in DNNs to reduce the model size and to accelerate inference [12]. However, quantisation is usually applied to all parameters of the DNN, which incurs an accuracy loss. Furthermore, quantised models may require dedicated hardware to carry out inference at a lower precision. Knowledge distillation transfers training knowledge from a more extensive teacher to a smaller student model [11]. The student model achieves similar accuracy to the teacher model and is spatially smaller. However, knowledge distillation is not easily automated to serve various model architectures and produces only a single student model rather than a portfolio of models suited for different operational conditions, such as specific memory budgets. Therefore, knowledge distillation does not scale for the varying resource requirements of deployments seen in heterogeneous edge environments.

2.5. Addressing open gaps with our contribution

Although existing compression methods have a range of benefits, they present one or more significant limitations that prohibit their use for on-demand deployment of production quality DNNs to edge devices. *DNNShifter leverages the accuracy-preserving benefits of unstructured pruning with the runtime performance improvements of structured pruning across various model sizes to suit different operational conditions seen in edge environments. This combination has not been previously explored in the literature.* `DNNShifter` creates an efficient training, pruning, and inference pipeline, which is highlighted in comparison to other DNN compression methods in Table 1. The `DNNShifter` framework and

Table 1

Comparison of Neural Architecture Search (NAS), Unstructured Pruning (UP), Structured Pruning (SP), Dynamic DNNs (D-D), and the proposed framework, `DNNShifter`.

	NAS	UP	SP	D-D	DNNShifter
Generates model variants to suit different operational conditions	✓	✗	✗	✓	✓
Produces memory efficient models	✗	✗	✓	✗	✓
Produces high accuracy models	✓	✓	✗	✓	✓
Produces compute efficient models	✗	✗	✓	✗	✓
On-demand switching of models at runtime	✗	✗	✗	✓	✓

the models generated by the framework meet the requirements for deploying DNNs in edge environments. However, existing methods have one or more limitations making them less suited for edge systems. The next section explores the underlying methodology and implementation of `DNNShifter`.

3. DNNShifter

`DNNShifter` is a framework that can be employed for resource-constrained environments, such as at the network edge or the extreme edge that has relatively limited computational capabilities. The framework prunes production quality DNN models on-demand and provides automated runtime model switching for inference on constrained resources. `DNNShifter` can be employed by system administrators for managing the life cycle of ML application development, deployment, and simulation environments to address the following challenges:

Rapidly obtaining production quality models: In real-time, `DNNShifter` offers structured pruning of large sparse DNN models that cannot be run on hardware-limited resources. The framework derives pruned model variants for the target resource without a significant accuracy loss while achieving this on a small monetary, computation, and energy budget. This contrasts existing approaches that employ NAS [7] or parameter fine-tuning [29].

Hardware agnostic automated model pruning: `DNNShifter` creates a portfolio of hardware-independent pruned model variants with different performance characteristics (e.g. model size, speed, and accuracy) to suit all deployment conditions. The model variants can be deployed across different resource tiers based on operational conditions, such as resource availability and performance targets. The approach adopted by `DNNShifter` is hardware agnostic and is not specific to specialised hardware such as a GPU [25].

Real-time model switching at runtime: Once a model portfolio has been deployed on the target hardware, `DNNShifter` utilises the portfolio of pruned model variants to select a model variant suited for a given operational condition. The framework facilitates the adaptation of the model to suit variations in the operational conditions with low overheads. The underlying method in `DNNShifter` switches the active model for inference from the portfolio via inflation (which activates the pruned model) and deflation (which further compresses and deactivates the pruned model) to match operational demand.

`DNNShifter` is envisioned to be a holistic end-to-end solution for ML edge applications that reduces human administrator interventions or domain-specific knowledge for creating pruned models. `DNNShifter` can also benchmark different pruning algorithms on heterogeneous hardware and make informed decisions in the life cycle of edge-based ML applications.

This section will further present the observations that motivated the design of `DNNShifter` and provides an overview of the framework.
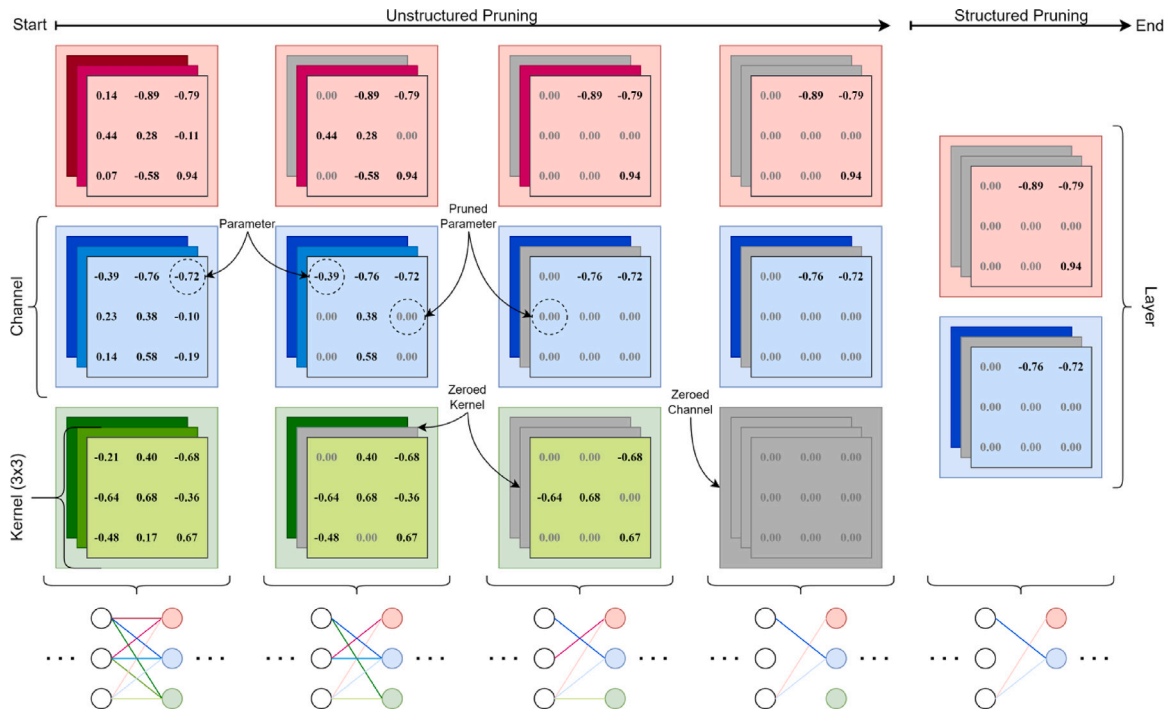
Fig. 2. Structured pruning zero-valued data structures obtained from unstructured pruning.

3.1. Motivation

A variety of model pruning methods have been presented in the literature for reducing the complexity of production models to suit resource-constrained environments while maintaining accuracy [20]. Traditional pruning methods are limited in multiple ways: (a) many require further time-consuming fine-tuning after the initial pruned model is obtained [29], (b) many rely on hardware accelerators [25], and (c) pruning often requires a costly trial and error process to determine the optimal pruned model for a given target resources [32]. Current pruning methods are unsuitable for real-time execution in critical scenarios, such as on-device video analytics, that require sub-second latency to preserve optimal service quality [35].

DNNShifter was developed to address the above limitations by leveraging the following two observations:

3.1.1. Aggregating and pruning unstructured sparsity

During unstructured pruning, aggregating parameters from sparse data structures will result in fully prunable data structures that directly impact the model size and inference latency. Fig. 2 highlights this observation. During unstructured pruning, the parameters of a convolutional kernel are set to zero values. The data structures (matrices) representing the kernels may be sparse (not all values are zero) and, therefore, cannot be pruned without compromising accuracy (shown as unprunable data structure). Zero matrices are obtained as pruning progresses by using the parameter ranking algorithm used in unstructured pruning. A data structure that has full zero values is prunable and, by employing structured pruning, can be removed from the model. This results in reducing the model size and, thereby, inference latency.

DNNShifter *leverages this observation to prune sparse models using structural pruning without degrading model accuracy. Since model accuracy is preserved,* DNNShifter *does not require fine-tuning after pruning.*

3.1.2. Further compression of remaining model sparsity

During runtime, inactive models from the portfolio can be further compressed to reduce overheads. After structured pruning, the remaining unprunable data structures contain sparse matrices (Fig. 2).

Sparse matrices have repeating and compressible data patterns (of zeroed weights). Therefore, the model can be encoded (deflated) into smaller representations while the model is inactive. For example, such deflation may be applied when downloading the model portfolio from a cloud/edge server to target device hardware or, during runtime, to models in a portfolio that are not actively inferring.

DNNShifter *uses this observation to load the entire portfolio of deflated models into memory during runtime. When a specific model is required for inference, it is decoded (inflated) in (near) real-time. This allows model switching in response to varying operational conditions on the edge and is significantly faster than existing methods.*

3.2. Framework overview

This section presents an overview of the DNNShifter framework. It operates in three phases, as shown in Fig. 3:

Phase 1: Offline training of production quality DNNs with unstructured ranking - In this phase, model training and parameter ranking are combined into a single iterative training process. A production-quality DNN model is taken as input by an unstructured pruning method. Then, the insignificant parameters of the model are masked between each training iteration by an unstructured ranking method to produce a portfolio of model variants (one per iteration) with sparse data structures (referred to as sparse models).

Phase 2: On-demand conversion from sparse models to pruned models - The portfolio of sparse models is pruned via structured pruning to obtain pruned model variants that can be deployed on a target hardware resource.

Phase 3: Runtime model switching - The portfolio is deployed, monitored, and adapts to varying operational conditions by switching the active pruned model variant at runtime.

Phase 1 builds on an existing technique and Phase 2 and Phase 3 comprise nine modules. The next sections discuss these.

3.3. Phase 1: Model training and parameter ranking

This phase uses an unstructured ranking algorithm to produce highly sparse models from production-quality DNNs (dense models)
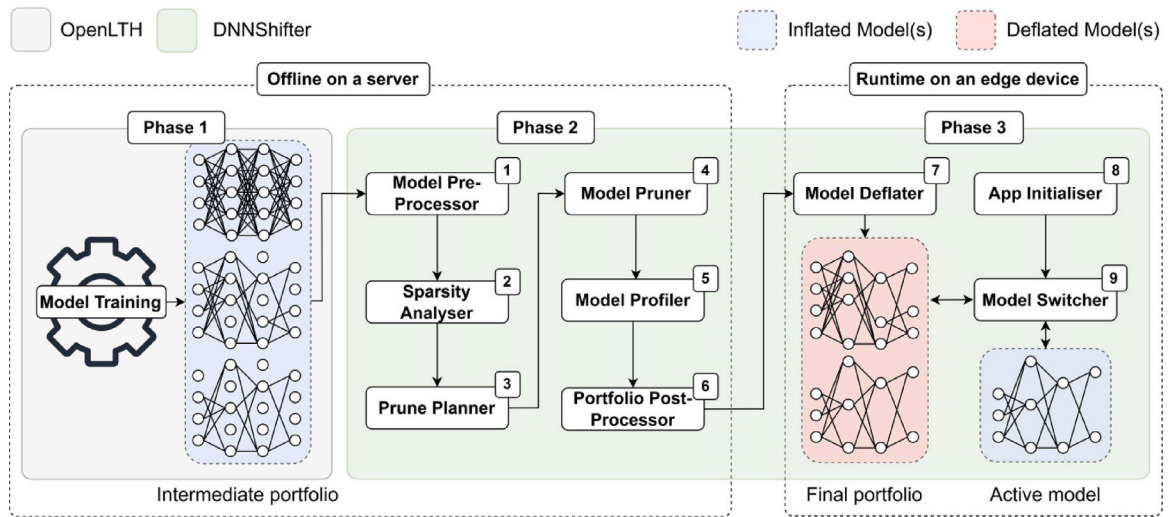
Fig. 3. Overview of the `DNNShifter` framework.

while maintaining viable model accuracy. It is to be noted that the sparse models obtained from this phase will include zero values in the model parameters. However, since they are not removed from the data structures until the next phase, sparse models are not smaller in size than the dense model. Choosing an unstructured ranking algorithm over structured pruning eliminates the need for fine-tuning after training to recover model accuracy [13,29]. In addition, such a ranking approach between training iterations has two advantages.

Firstly, `DNNShifter` simplifies model parameter ranking so that a user does not require expert knowledge of ranking algorithms and no additional hyperparameters need be configured.

Secondly, `DNNShifter` improves the model pruning pipeline efficiency. A conventional pruning pipeline consists of training the model, compressing using structured pruning methods, profiling, and iteratively fine-tuning the pruned model for the target hardware. While training and compression can occur offline on large-scale computational resources, fine-tuning will need to be carried out on the target hardware that may be resource-limited. The final accuracy of the model cannot be determined until this time-consuming pipeline is completed. If the desired accuracy is not obtained, then the entire sequence of the pruning pipeline must be repeated with a different set of pruning hyperparameters. In addition, only a single pruned model will be obtained at the end of the pipeline that meets specific operational conditions. If the operational condition changes, the entire pruning pipeline must be repeated.

`DNNShifter` improves the efficiency of the pruning pipeline in three ways by integrating ranking within the training iterations: (i) The final model accuracy that can be achieved is known before the pipeline completes. Therefore, the pipeline can be reinitialised in advance with new hyperparameters if the target accuracy cannot be achieved. (ii) Fine-tuning, a computationally intensive task, is eliminated on the target hardware resource that may be relatively resource constrained. Therefore, rapid and on-demand deployments of DNN models are feasible since fine-tuning does not need to be carried out. (iii) A portfolio of pruned models can be generated that will suit a range of operational conditions on the target hardware resource by running the pruning pipeline once. This allows for adapting a model that is deployed at runtime.

`DNNShifter` implements a modified version of the Open Lottery Ticket Hypothesis (OpenLTH) framework.[1] No modifications were made to the training process. Instead, `DNNShifter` adds the structured pruning and model switching phases (Phase 2 and Phase 3)
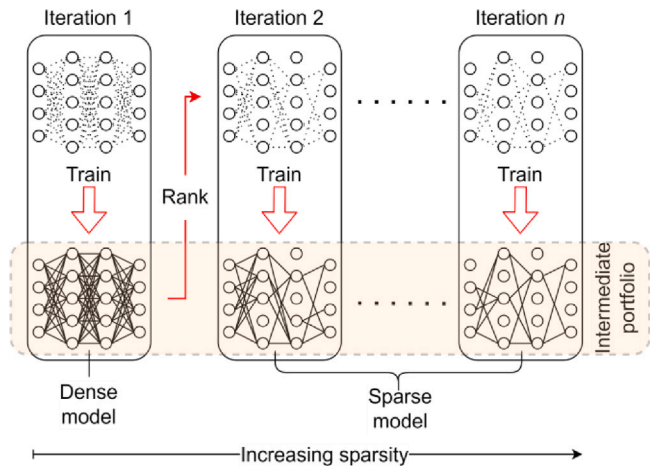
---

[1] https://github.com/facebookresearch/open_lth

Fig. 4. The unstructured pruning method incorporated in `DNNShifter` uses the combined approach of repetitive training and model ranking between training iterations.

which will be discussed later. The Lottery Ticket Hypothesis (LTH) articulates that highly accurate sparse models can be obtained from dense models [23] (shown in Fig. 4) and is underpinned by Iterative Magnitude Pruning (IMP) with weight rewinding [36] method that `DNNShifter` also employs. IMP with rewinding is chosen since it performs well across all available models and datasets. Alternatives, such as SynFlow [27], only perform well for specific models or datasets [26].

Fig. 4 illustrates Phase 1 of `DNNShifter` to generate sparse models using LTH. The model is trained and then ranked by the IMP algorithm in each iteration. The resulting sparse model from each iteration is saved into an intermediate portfolio of sparse models that will be pruned in the next phase. The sparse model from each iteration is provided as input for the next iteration. Model sparsity increases with the number of iterations up to a user-defined limit.

3.4. Phase 2: Converting sparse models to pruned models

In this phase, the n sparse models from the intermediate portfolio are converted into m pruned models using structured pruning. This phase consists of six processing modules that pre-process sparse models, identify prunable data structures within each sparse model, generates plans for pruning, and then use structured pruning to generate pruned

---

Algorithm 1: `DNNShifter` Prune Planning

Data: Set of all zero channels C_{Zero},
Convolutional Layer L_n
Result: Prunable channels (C_{in}, C_{out}) in L_n
1   $C_{in} \leftarrow \{\}$
2   $C_{out} \leftarrow \{ c_{zero} \mid c_{zero} \in (L_n \cap C_{Zero})\}$
3   **if** *(n > 0)* **then**
4 | $C_{in} \leftarrow \{ c_{zero} \mid c_{zero} \in (L_{n-1} \cap C_{Zero})\}$
5   **return** (C_{in}, C_{out})

Algorithm 2: `DNNShifter` Batch Pruning

Data: Prunable channel indices (C_{in}, C_{out}) in L_n
Result: Pruned convolutional layer L'_n
1   $|C'_{in}| \leftarrow |L_n(C_{in})| - |C_{in}|$
2   $|C'_{out}| \leftarrow |L_n(C_{out})| - |C_{out}|$
3   $L'_n \leftarrow$ create new layer$(|C'_{in}|, |C'_{out}|)$
4   $L'_n(C_{out}) \leftarrow L_n(C_{out}) \setminus C_{out}$
5   $L'_n(bias) \leftarrow L_n(bias) \setminus C_{out}$
6   **if** *(n > 0)* **then**
7 | $L'_n(C_{in}) \leftarrow L_n(C_{in}) \setminus C_{in}$
8   **return** L'_n

---

models. Each sparse model is processed from the intermediate portfolio to produce a final portfolio. Each module of this phase is detailed below:

Module 1 - Model Pre-Processor

This pre-processing module simplifies the DNN model architecture by fusing the convolution and batch normalisation layers [37]. This fusion removes batch normalisation parameters, thereby reducing the complexity of generating a pruning plan for the model, as only convolutional layers and their dependants must be considered (further discussed in Module 3).

Module 2 - Sparsity Analyser

This module builds on the method illustrated in Fig. 2 that identifies convolutional kernels with entirely zero values. However, these kernels cannot be removed without further planning since a DNN's architecture does not naturally lend itself to the removal of kernels alone. Instead, a kernel can be removed if all kernels in a channel can be removed. To this end, channels that have all kernels with zero values are further indexed to create prunable convolutional channels.

Module 3 - Prune Planner

In the existing literature, convolution channels are removed from a model iteratively to minimise accuracy loss. However, this is inefficient for two reasons. Firstly, pruning channels is computationally intensive since entire convolutional layers comprising large multi-dimensional parameter arrays will be rebuilt when prunable channels are removed. Secondly, each prunable channel depends on the channel of the next convolutional layer, which will also be rebuilt. Therefore, pruning sequentially incurs overheads. `DNNShifter` breaks this dependency and removes all prunable channels at the same time. This is achieved by the prune planner module, which creates a concurrent data structure of prunable channel indices.

This module uses Algorithm 1 where each zero channel c_{zero} (channels with all weights set to zero) in a set of all zero channels C_{Zero} (indexed in Module 2) is mapped to a convolutional layer L_n where $0 \leq n < D_{conv}$ (model convolutional layer depth). Each convolutional layer receives two sets of zero channels. The first set, C_{in} is the set of prunable *out* channels from the previous convolutional layer L_{n-1}: these indices correspond to the prunable *in* channels of L_n. The second set, C_{out}, is the prunable *out* channels of L_n. When $n = 0$ (first convolutional layer), there is no C_{in}. Therefore this layer receives an empty set. The returned prune plan $((C_{in}, C_{out}))$ contains all zero channels that are to be pruned in Module 4 for a given convolutional layer.

Module 4 - Model Pruner

The pruning plan from Module 3 is used to prune a sparse model from the intermediate portfolio in real time. This module executes the pruning plan by rebuilding each convolutional layer without the prunable channels and the biases of the channels. As all prunable channels are made available from Module 3, this module prunes all in/out channels in a single batch operation, significantly reducing computational overhead and enabling real-time pruning. After prune planning, this module is executed in parallel to concurrently prune each

convolutional layer, forming a series of pruned layers L' that replaces the original unpruned layers L.

This module uses in Algorithm 2 where a pruned layer L'_n is created with the smaller channel size $|C'_{in}|$ and $|C'_{out}|$ (Lines 1-3). Afterwards, the pruned set of remaining channels and bias are transferred from L_n to L'_n (Lines 4–7).

Module 5 - Model Profiler

This module benchmarks the pruned model to obtain metrics: accuracy, inference latency, model size, and the maximum memory required to run the model. This is achieved using a test dataset. The metrics relevant to each model are stored as metadata for the next module for selecting a suitable pruned model from a portfolio.

Module 6 - Portfolio Post-Processor

A portfolio of n pruned models is generated. This module refines the portfolio to eventually only include m pruned models ($m \leq n$) with distinct performance characteristics (pruned models with similar characteristics are removed).

3.5. Phase 3: Further compression and model switching

A portfolio of production-quality DNN models is trained in the first phase and then compressed via structured pruning in the second phase. In this third phase, models are further compressed while not being used (inactive). `DNNShifter` encodes the portfolio of pruned models into a significantly smaller package before deploying it to the storage of the target device using the lossless DEFLATE [38] compression algorithm. On application initialisation, `DNNShifter` loads the entire portfolio into memory, and then one model is activated (inflated) to enable inference. Encoding models in this manner (deflating) is possible since zero weights repeat in highly sparse DNN models obtained from training. However, out-of-the-box production quality models are dense (most of their weights are not set to zero). Therefore, applying this compression algorithm to dense models will not provide any benefit. Each module of this phase is detailed below:

Module 7 - Model Deflater

This module sorts the model portfolio by model size (a proxy for performance characteristics), then shrinks the entire model portfolio into a smaller, sorted, and easily deployable package using DEFLATE before it is transferred to the target devices.

Module 8 - Application Initialser

This module loads the entire portfolio of deflated models into device memory. First, it selects a model with the median model size. Then, this model is decompressed in memory to enable application inference (we refer to this as inflation). Note that the inflated model is a pruned model variant from Phase 2. It is smaller and faster for inference than an equivalent dense model (Fig. 6).

Module 9 - Model Switcher

The available memory and CPU load may vary due to the number of running applications and the workload of each application on the device. For example, inference performance metrics, such as queries per second (QPS), may vary over time for an application [39]. During a low load on the edge device, a higher QPS can be achieved, during which time a larger model from the portfolio can be decompressed in the device memory (inflation); the larger model will improve inference accuracy. Inversely, a decreasing QPS suggests a high load and a smaller model from the portfolio is inflated to improve inference latency performance. DNNShifter does not require searching the entire portfolio to switch between models. Instead, this module selects the next or previous model from the portfolio depending on the QPS trend. Therefore, model switching can be rapidly obtained with minimum overheads.

4. Experiments

This section first presents the experimental testbed and baseline models in Section 4.1 and then considers three key aspects of DNNShifter:

(1) The time to generate a portfolio of models for addressing Challenge 1 (Phase 1 of DNNShifter). We will compare against state-of-the-art NAS methods, namely DARTS [16], RepVGG [14], and Pre-NAS [17], for evaluating this. We will demonstrate in Section 4.2 that DNNShifter will generate a portfolio faster and more efficiently.

(2) The accuracy achieved and the time taken for inference by the models for addressing Challenge 2 (Phase 2 of DNNShifter). Two categories of pruning algorithms, unstructured and structured, are considered here. The unstructured pruning algorithms considered are random pruning, magnitude pruning [19], SynFlow [27], and NTK-SAP [22]. The structured pruning algorithms considered include similarities-aware [40], l^1 norm [40], EasiEdge [15], and ProsPr [30]. We will demonstrate in Sections 4.3.1 and 4.3.2 that DNNShifter obtains better accuracy and an improved inference speedup for the pruned models compared to unstructured pruning methods. It will also be demonstrated in Section 4.3.3 that when compared to structured pruning methods, the pruned models obtained from DNNShifter have better accuracy for a desired model size. We will show in Section 4.3.4 that DNNShifter has overheads that are multiple magnitudes lower than structured pruning methods.

(3) The overheads for dynamically switching a model in memory for addressing Challenge 3 (Phase 3 of DNNShifter). We will demonstrate in Section 4.4 that compared to model switching approaches, such as Model Ensemble [41] and Dynamic once-for-all (Dynamic-OFA) [10], DNNShifter has lower model switching overheads and memory utilisation.

4.1. Experimental setup

Two production DNN models trained on the CIFAR-10 [42] and Tiny ImageNet [43] dataset are considered. First is VGG-16 [2] trained on CIFAR-10, which represents a feedforward DNN model. Second is ResNet-50 [44] trained on Tiny ImageNet, which is a more complex branching DNN model.

Table 2 presents the baseline results and hyperparameters:

Models, Datasets, and Hyperparameters - VGG-16 is the OpenLTH configuration that has one linear layer [23], and ResNet-50 is the default ImageNet configuration [44]. CIFAR-10 consists of 50,000 training images and 10,000 test images divided equally across 10 classes. Tiny ImageNet is a subset of ImageNet consisting of 100,000 training images and 10,000 test images divided equally across 200 classes. Tiny ImageNet results are reported for both Top-1 and Top-5 as recommended by model pruning literature [20]. The baseline results

Table 2
Baseline results and training hyperparameters for a production quality dense VGG-16 and ResNet-50 models.

	VGG-16	ResNet-50
Dataset	CIFAR-10	Tiny ImageNet
Params. (M)	14.72	25.56
Size (MB)	56.2	97.8
Top-1 Acc. (%)	93.21 ± 0.06	54.99 ± 0.14
Epochs	160	200
Batch Size	128	256
Learning Rate	0.1	0.2
Milestone Steps[a]	80, 120	100, 150

[a] Learning rate drops by a factor of gamma, 0.1, at each milestone step. Both models use the SGD optimiser, momentum of 0.9 and weight decay of 0.0001.

were obtained using the training routine from OpenLTH[2] as defined in Section 3.2.

Testbed - We use an AMD EPYC 7713P 64-core CPU and Nvidia RTX A6000 GPU to train the models, as such resources are representative of those in a cloud data centre. Model inference and runtime switching is carried out with an Intel i7-9750H 6-core CPU and an Nvidia RTX 2080 (Max-Q) GPU comparable to an edge server that may be used in a production setting.

Trial Counts and Reporting Methods - All DNN training methodologies and experiments were conducted a minimum of three times, except for those in Section 4.2. In Section 4.2, each NAS approach was executed only once due to computational and time constraints. Unless otherwise specified, model performance indicators like accuracy, memory usage, and latency are presented in tables and figures as the mean from all trials accompanied by confidence intervals spanning one standard deviation. In addition, where possible, experiments are carried out across 8 different compression ratios (2, 4, 8, 16, 32, 64, 128, 256).

4.2. Model training and portfolio generation (Phase 1)

This study will demonstrate that DNNShifter will generate a portfolio of models from a base architecture with a higher search efficiency than comparable NAS methods. Search efficiency is the percentage of optimal model variants in the portfolio over the total number of searched variants. An optimal model variant is one that is not outperformed on all performance metrics by another variant and is obtained using Pareto optimality. The performance metrics considered in this article are model size, inference latency, and model accuracy. For example, training a single model that reaches an adequate accuracy has a search efficiency of 100%. However, if training occurs N times, then the model with the highest accuracy from the N training rounds is optimal, but search efficiency drops to $100/N\%$.

DNNShifter creates a model portfolio by iteratively pruning the largest model variant into progressively smaller variants (discussed in Section 3.3). Each pruning iteration is equivalent to searching the model architecture once for a variant. We compare DNNShifter against three different NAS-based methods, which search a model architecture for optimal model variants.

The first is DARTS, an accelerated NAS method that generates model variants from a continuous search space via gradient descent. Compared to older NAS methods such as NasNet, DARTS is 500x faster. In addition, DARTS is a NAS approach that automatically generates a portfolio of models.

The second is RepVGG, which employs a family of VGG-like model variants from a set of discrete hyperparameters that scale various model architecture properties. In total, 228 model variants are individually trained to identify the optimal set of model variants based on hyperparameters presented in RepVGG literature [14].

---

[2] Using Python 3.8.10, torch 1.13.0+cu116, and torchvision 0.14.0+cu116.

Table 3

Comparing the search efficiency of DNNShifter against NAS methods when generating a portfolio of model variants. Mean Top-1 accuracy and standard deviation of the optimal model variant portfolio is recorded for the CIFAR-10 dataset.

	# Model Variants	Model Params. (M)			Params. Trained (M)	GPU-days	Acc. (%)	# Optimal Variants	Search Efficiency (%)
		Min.	Avg.	Max.					
PreNAS [17]	1770	41.44	77.64	102.78	543.5	9.11	96.96 ± 1.48	7	0.39
RepVGG [14]	228	8.30	48.81	110.96	12 329.0	26.09	94.96 ± 0.16	13	5.70
DARTS [16]	50	3.30	3.30	3.30	165.0	1.77	74.01 ± 16.9	3	8.00
DNNShifter	9	0.05	3.32	14.72	132.2	0.28	93.25 ± 0.66	4	44.44

The third is PreNAS, a modern NAS that generates models using the emerging vision transformer model architecture [45]. PreNAS is a one-shot NAS that decides on a set of optimal model variants and only trains those candidates, significantly reducing computational requirements.

DNNShifter has one hyperparameter, n, which specifies how many model variants should be generated where each variant is twice as compressed as the previous. For example, $n = 8$ generates a portfolio of model variants up to the compression ratio 256 (2^n). The first variant is the original dense model with no sparsity.

Table 3 contrasts DNNShifter against DARTS, RepVGG, and PreNAS. DNNShifter generates 4 optimal model variants out of a portfolio of 9, resulting in a search efficiency of 44.44%. This is more efficient than the NAS-based methods. The number of parameters trained using DARTS is divided across a more extensive portfolio of models. These models are not sufficiently diverse, resulting in a low search efficiency of DARTS. The DARTS search method requires over 6 h to create the portfolio. Then, each variant is individually trained and evaluated, totalling a training time that is 6x longer than DNNShifter. RepVGG and PreNAS achieve a higher model accuracy than DNNShifter, but each model variant is up to one order of magnitude larger in parameter count. As this study evaluates training time as a function of parameter count, the trend that is seen in Table 3 generalises for all model architectures of different sizes.

> **Observation 1:** The DNNShifter method for generating a portfolio of models via iterative pruning is more resource and time-efficient than NAS-based methods.

4.3. Performance of sparse and pruned models (Phase 2)

This study will demonstrate that DNNShifter produces pruned models of the same or better accuracy than other unstructured (Section 4.3.1) and structured (Section 4.3.3) pruning methods. In addition, it is demonstrated that the pruned models generated by DNNShifter are smaller and faster than sparse models, which is quantified for various compression ratios (Section 4.3.2).

4.3.1. Comparing accuracy against sparse models

We will first contrast the choice of the unstructured pruning method in DNNShifter against other unstructured pruning methods. Unstructured pruning methods produce a sparse model variant where parameters are set to zero. In this paper, a sparse model with a compression ratio of C has for every C parameters, $C - 1$ parameters set to zero; this is also presented in the literature [20,27]. As seen in Section 3.3, DNNShifter utilises IMP with rewinding as its unstructured pruning method. This study evaluates DNNShifter against random pruning, which is a naive method, magnitude pruning, SynFlow, and NTK-SAP. For each method, the baseline models in Table 2 are iteratively pruned up to eight times where a compression ratio of 2^n is achieved per iteration n as described in Section 4.2.

Fig. 5 shows the change in test accuracy as the compression ratio increases for VGG-16 and Resnet-50. For all methods, accuracy decreases as the compression ratio increases. However, the rate of decline varies per method, whereas DNNShifter maintains the highest accuracy

in all cases. For example, for VGG-16 on CIFAR-10 at a compression ratio of 256, DNNShifter accuracy dropped by 1.57%, whereas SynFlow, NTK-SAP, and Random pruning dropped by 6.21%, 5.15%, and 31.09%, respectively. Magnitude pruning compromises almost all of its accuracy by a compression ratio of 256. Magnitude pruning causes the model to become unusable at high compression ratios due to layer collapse (when an entire layer is pruned) [27]. However, SynFlow was designed to avoid layer collapse and maintain useable accuracy at high compression ratios. DNNShifter also does not encounter layer collapse as it uses the rewinding approach from IMP that stabilises the sparse model [36].

For ResNet-50 on Tiny ImageNet, for top-1 and top-5 test accuracy, DNNShifter accuracy reduction is less than half of SynFlow at higher compression ratios. In contrast, magnitude pruning, and NTK-SAP undergo layer collapse after a compression ratio of 32, and 64, respectively.

> **Observation 2:** DNNShifter preserves the highest accuracy for sparse models compared to existing methods. Furthermore, this enables DNNShifter to generate sparse models at extreme compression ratios, providing the opportunity for structured pruning.

The benefits of using structured pruning at extreme model sparsities are explored in the next subsection.

4.3.2. Comparing runtime performance against sparse models

Unstructured pruning methods do not provide inference acceleration or spatially reduce the model size at runtime. This is because the parameters are not spatially pruned but rather are merely made zero values in the model. DNNShifter removes pruned parameters via structured pruning. This study highlights the performance benefits of using pruned models from DNNShifter compared to the sparse models generated by unstructured pruning methods.

Fig. 6 shows the CPU and GPU inference speed up and spatial compression achieved for increasing compression ratios. For DNNShifter, inference speed-up is defined as the inference latency of the baseline model over the pruned model for a given compression ratio. For the other unstructured pruning methods, we consider inference speed-up as the inference latency of the baseline model over the sparse model for a given compression ratio. Similarly, spatial compression is the in-memory size of the baseline model over that of the in-memory size of the pruned model (for DNNShifter) or of the sparse model (for unstructured pruning methods) for a given compression ratio. For both metrics, DNNShifter provides improvements for all compression ratios, whereas unstructured pruning methods alone provide little or, in some cases, worse performance than the baseline model.

DNNShifter achieves up to 1.67x and 1.32x for VGG-16 and ResNet-50 on CPU and GPU, respectively, at no cost to the accuracy of the sparse model. The other unstructured pruning methods achieve a small speedup. However, the speedup varies due to irregular memory access in sparse models [46]. DNNShifter spatially prunes the sparse parameters and thus is not affected by structural irregularity. As the compression ratio increases, more sparse parameters are removed, resulting in a smaller model with lower CPU and GPU inference times.
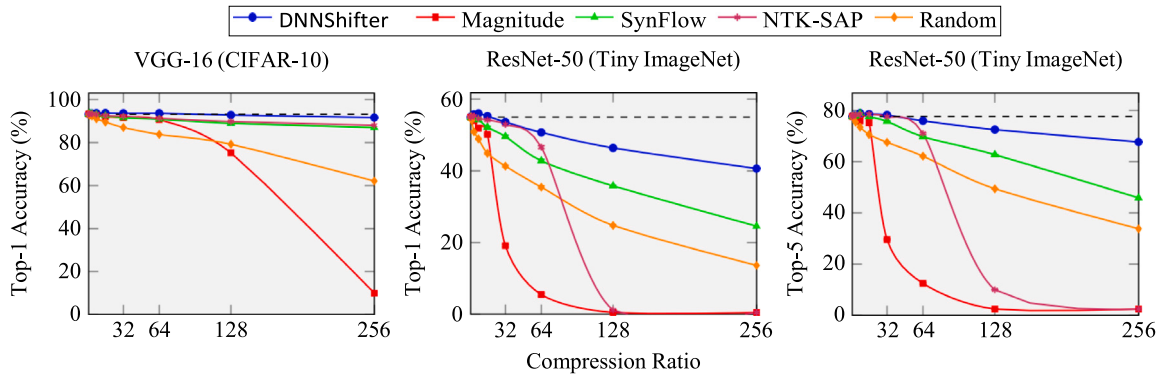
Fig. 5. Accuracy of unstructured pruning in `DNNShifter` against other methods as compression ratio increases; dashed line is baseline model accuracy.
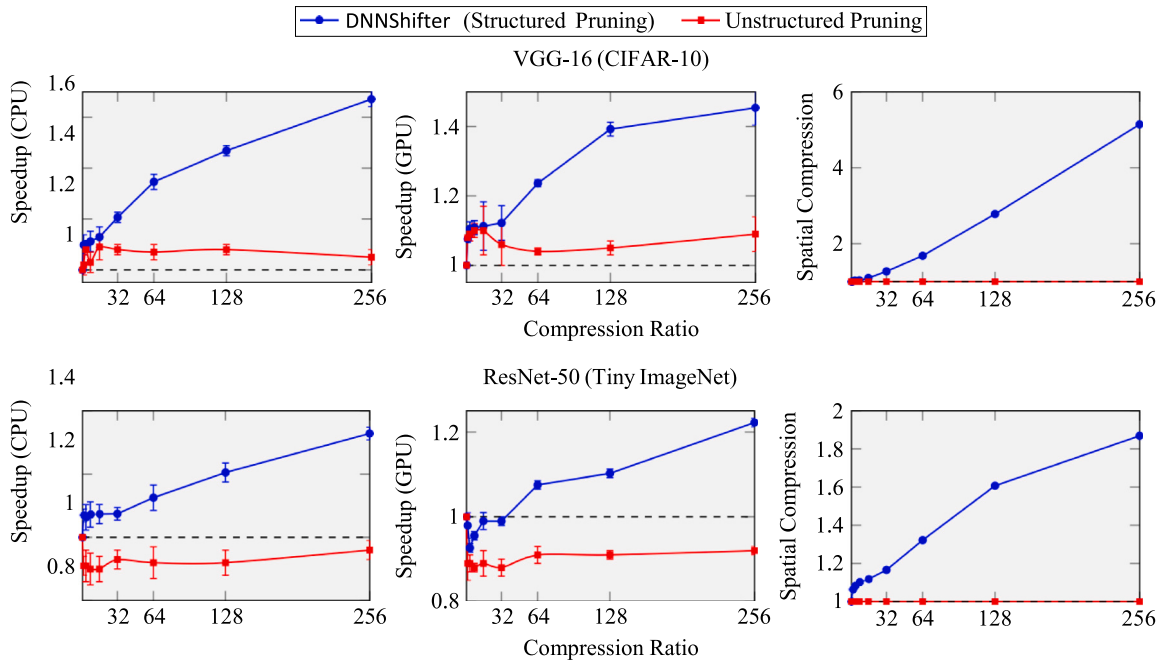

Fig. 6. Performance of `DNNShifter` against other unstructured pruning methods as compression ratio increases; dashed line is baseline model performance. Each plot is the mean of five runs with confidence intervals of one standard deviation.

`DNNShifter` at a compression ratio of 256 achieves a spatial compression on the sparse model of 5.14x and 1.87x for VGG-16 and ResNet-50, respectively. ResNet-50 has a lower spatial compression ratio as DNNShifter only removes linear connections using structural pruning. As such, any skip connections or downsampling layers in ResNet-50 are not pruned as it will impact model accuracy [47].

> **Observation 3:** `DNNShifter` reduces inference latency and sparse model sizes in memory without losing accuracy. This is in contrast to sparse models obtained from unstructured pruning that are unsuitable for edge environments since they have poor inference performance and no spatial compression.

4.3.3. Comparing accuracy against pruned models

This study contrasts `DNNShifter` against other structured pruning methods. Specifically, `DNNShifter` is demonstrated to have comparable accuracy to the original model while producing similarly sized or smaller pruned models than other structured pruning methods. `DNNShifter` is compared against RepVGG, EasiEdge, ProsPr, and two classic structured pruning methods: similarities-aware and l^1 norm.

RepVGG, as described in Section 4.2, creates VGG-like architectures by re-parameterising ResNet. In this study, RepVGG-A0 is the pruned VGG-16 model obtained from the baseline RepVGG-B0 [14]. EasiEdge is a recent structured pruning method that creates pruned models for edge deployment. ProsPr is another modern pruning method that learns which weights to prune within the first few steps of optimisation [30]. In this study, we use the structured pruning variation that prunes channels. Pruning using l^1 norm is a classic structured pruning method that ranks the importance of each channel using l^1 norm and then prunes the lowest value channels [40]. Similarities-aware is another classic structured pruning method that removes channels with similar outputs [40].

Table 4 shows the accuracy change of the pruned model and the total parameter count after pruning the baseline VGG-16 on CIFAR-10 using different structured pruning methods. The table is organised in descending order of parameter count, where the baseline VGG-16 models are considered first and increasingly pruned models towards the bottom. EasiEdge and ProsPr models are denoted using a prune degree as a percentage. Prune degree is the percentage of pruned parameters from the baseline. For example, EasiEdge-25% prunes VGG-16 by 25%. `DNNShifter` models are denoted using the compression ratio. For example, DNNShifter-2x has a compression ratio of 2, equivalent to a prune degree of 50%.

Table 4
Accuracy change and parameters of pruned VGG-16 models on CIFAR-10 for different structured pruning methods. Values in bold are the best results for each category. Underlined values are second best for each category.

	Acc. Change (%)	Params. (M)
VGG-16 (Baseline)	–	14.72
VGG-16 (Sim.-aware)	−5.82	11.04
VGG-16 (l^1 norm)	−5.14	9.57
RepVGG-A0	−0.40	8.30
EasiEdge-25%	+0.02	8.29
l^1 norm + Sim.-aware	−5.30	8.10
DNNShifter-2x	+0.18	7.54
ProsPr-80%	+0.01	4.01
DNNShifter-4x	+0.28	3.86
EasiEdge-50%	+0.13	3.68
ProsPr-90%	+0.04	2.00
DNNShifter-16x	+0.40	1.02
ProsPr-95%	−0.28	1.00
EasiEdge-80%	−0.22	0.76
EasiEdge-85%	−0.51	<u>0.46</u>
DNNShifter-64x	<u>+0.33</u>	**0.21**

Both classic structured pruning methods showed more than 5% accuracy reduction with a pruning degree of 35% or less. Combining the two methods allows for a similar accuracy loss but up to a pruning degree of 45%. RepVGG-A0 achieves the same pruning degree as the combined classic methods while only dropping 0.4% model accuracy. However, RepVGG does not have a smaller model variant than RepVGG-A0. DNNShifter and EasiEdge produce smaller models with better accuracy than the baseline model. DNNShifter-16x has the best accuracy improvement with a 0.4% gain, where a similarly sized EasiEdge-80% lost 0.22% accuracy. The smallest EasiEdge variant, namely EasiEdge-85%, with 0.46M parameters, 0.51% loss in accuracy, whereas DNNShifter-64x is over twice as small, with 0.21M parameters and gains 0.33% accuracy. ProsPr maintains a positive accuracy change up until models of size 2M parameters; however, accuracy remains lower than DNNShifter at all model sizes.

> **Observation 4:** DNNShifter produces smaller and more accurate pruned models than other structured pruning methods.

4.3.4. Pruning time against structured pruning methods

Fig. 7 shows the pruning time in seconds of various structured pruning methods to prune a ResNet model. l^1 norm requires almost 3,923 s to prune and fine-tune the model. EasiEdge does not require fine-tuning, but the ranking process it employs using Taylor approximations is exhaustive, thus requiring 4,740 s. RepVGG does not require ranking. Instead, it re-parameterises the model, which only requires 8 s. Although this is a relatively small cost, an 8-second overhead per training round may equate to a substantial overhead for certain use cases. For example, consider pruning during the rounds of federated learning [48]. NTK-SAP requires 20 epochs of pre-training to generate an unstructured pruning mask resulting in 544 s of overhead. DNNShifter prunes a model in a sub-second time frame. For ResNet, it is on an average of 120 ms, or less than 3 frames for a 30 frames/second real-time edge video analytics application [35], as opposed to tens of seconds to minutes of downtime with existing approaches.

> **Observation 5:** DNNShifter enables (near) real-time structured pruning of DNN models and is at least one order of magnitude faster than other structured pruning methods.
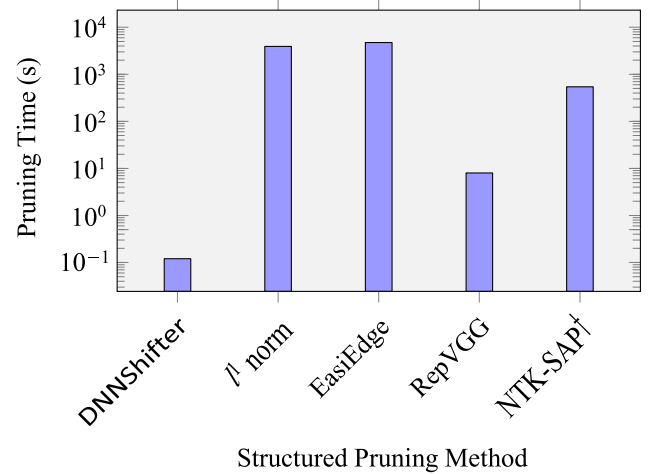

Fig. 7. Average pruning time of a ResNet model using structured pruning.†NTK-SAP unstructured pruning with DNNShifter Phase 2 structured pruning.

4.4. Performance of model switching (Phase 3)

Model switching enables an application to respond to changing runtime conditions by selecting a suitable model for inference from a pruned model portfolio. This study compares the in-memory compression and model switching method of DNNShifter against the classic methods, namely model ensemble and Dynamic-OFA. The model ensemble method hosts simultaneous models in memory, and Dynamic-OFA uses a smaller sub-network within a single DNN to match operational demands.

Table 5 compares runtime switching of DNNShifter against model ensemble and Dynamic-OFA. DNNShifter has a portfolio of four pruned VGG-16 models obtained in Section 4.2 with an accuracy range of 91.64–93.71%, model portfolio size of 30.4–66.1MB, and a CPU inference speedup of 1.20–1.67x. A model ensemble of the same four VGG-16 models and a Dynamic-OFA model are also noted. Memory utilisation is the size of the model portfolio in memory. DNNShifter's memory utilisation is variable since inactive models are further compressed in memory (Section 3). However, in the model ensemble method, all models are uncompressed and hosted in memory. Similarly, Dynamic-OFA maintains the entire model in memory, even though only a smaller sub-network may be used during inference. DNNShifter utilises as little as 3.8x less memory for its model portfolio compared to the model ensemble method.

Decision overhead is the wall clock time for a model-switching method to select a model from the portfolio. For example, the model ensemble method runs inference on all models in the portfolio and then chooses the output with the highest confidence. On the other hand, Dynamic-OFA selects one DNN configuration to run inference and then re-configures the DNN to that selection. DNNShifter inflates the appropriate model from in-memory and has an average decision overhead of 43 ms, which is up to 11.9x faster than Dynamic-OFA.

> **Observation 6:** DNNShifter provides model switching with lower memory and decision overheads compared to existing methods.

5. Discussion and conclusion

Deploying production-quality DNNs in resource-constrained environments is essential for facilitating edge machine learning. Model compression offers techniques to derive model variants from a production-quality model suited for resource-constrained environments. However,

Table 5
Memory utilisation and decision overhead of using model switching methods for VGG-16. Relative values to `DNNShifter` are recorded in parentheses.

	Mem. Util. (MB)	Overhead (ms)
Model Ensemble [41]	115.8 (1.8–3.8x)	49 (1.14x)
Dynamic-OFA [10]	56.2 (0.9–1.8x)	512 (11.9x)
`DNNShifter`	30.4–66.1	43

obtaining model variants that preserve accuracy and can be compressed to reduce the resource footprint and achieve low inference latencies is challenging. Moreover, existing research has limited focus on adapting model variants to changing runtime conditions.

`DNNShifter` addresses the above concerns by developing an end-to-end framework that incorporates a novel pruning method and a time and resource-efficient pipeline for model training, compression, and runtime switching. `DNNShifter` prepares model variants magnitudes faster than state-of-the-art neural architectural search, thus facilitating rapid and on-demand model deployments at the edge. The pruned model variants maintain the same accuracy as their production quality counterparts. They are suited for edge deployments since they are lightweight and adaptable to runtime conditions.

`DNNShifter` was designed to accommodate existing ML training and inference pipelines. `DNNShifter` does not introduce any extra hyperparameters or dependencies other than requiring a user-specified maximum portfolio size. The structured pruning method of `DNNShifter` can easily be used: (1) for one-time optimisation to pre-existing pre-trained DNN models, (2) in conjunction with the other phases to create a training and inference pipeline from scratch, or (3) in any combination of `DNNShifter`'s phases. Thus, `DNNShifter` is easily transferable to existing ML applications and products.

`DNNShifter` is primarily limited by the high computation cost of training sparse models. There is potential for structured pruning to be conducted at the initialisation of the model (before training) with minimal accuracy loss [49,50]. This will be explored in the future.

CRediT authorship contribution statement

Bailey J. Eccles: Conceptualization, Methodology, Software, Validation, Formal analysis, Writing – original draft. **Philip Rodgers:** Writing – review & editing. **Peter Kilpatrick:** Writing – review & editing. **Ivor Spence:** Writing – review & editing. **Blesson Varghese:** Conceptualization, Resources, Writing – review & editing, Supervision, Funding acquisition.

Declaration of competing interest

The authors declare the following financial interests/personal relationships which may be considered as potential competing interests: Bailey Eccles has patent #Edge-Masking Guided Node Pruning. PCT/US2022/053590 pending to Rakuten Mobile.

Data availability

Data will be made available on request.

Acknowledgements

References

[1] A. Krizhevsky, I. Sutskever, G.E. Hinton, ImageNet classification with deep convolutional neural networks, in: Advances in Neural Information Processing Systems, Vol. 25, 2012.

[2] K. Simonyan, A. Zisserman, Very deep convolutional networks for large-scale image recognition, in: 3rd International Conference on Learning Representations, 2015, pp. 1–14.

[3] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, L. Fei-Fei, ImageNet: A large-scale hierarchical image database, in: IEEE Conference on Computer Vision and Pattern Recognition, 2009.

[4] J. Turner, J. Cano, V. Radu, E.J. Crowley, M. O'Boyle, A. Storkey, Characterising across-stack optimisations for deep convolutional neural networks, in: 2018 IEEE International Symposium on Workload Characterization, IEEE Computer Society, 2018, pp. 101–110.

[5] S. Han, J. Pool, J. Tran, W.J. Dally, Learning both weights and connections for efficient neural networks, in: 28th International Conference on Neural Information Processing Systems, 2015, pp. 1135–1143.

[6] Z. Wang, T. Luo, R.S.M. Goh, J.T. Zhou, EDCompress: Energy-aware model compression for dataflows, IEEE Trans. Neural Netw. Learn. Syst. (2022) 1–13.

[7] H. Cai, C. Gan, T. Wang, Z. Zhang, S. Han, Once for all: Train one network and specialize it for efficient deployment, in: International Conference on Learning Representations, 2020.

[8] B. Zoph, V. Vasudevan, J. Shlens, Q.V. Le, Learning transferable architectures for scalable image recognition, in: IEEE/CVF Conference on Computer Vision and Pattern Recognition, 2018, pp. 8697–8710.

[9] J. Zhang, S. Elnikety, S. Zarar, A. Gupta, S. Garg, Model-Switching: Dealing with fluctuating workloads in Machine-Learning-as-a-Service systems, in: 12th USENIX Workshop on Hot Topics in Cloud Computing, 2020.

[10] W. Lou, L. Xun, M. Sabetsarvestani, J. Bi, J. Hare, G. Merrett, Dynamic-OFA: Runtime DNN architecture switching for performance scaling on heterogeneous embedded platforms, in: IEEE/CVF Conference on Computer Vision and Pattern Recognition Workshops, 2021, pp. 3104–3112.

[11] G. Hinton, O. Vinyals, J. Dean, et al., Distilling the knowledge in a neural network, 2015, arXiv preprint arXiv:1503.02531.

[12] U. Kulkarni, A.S. Hosamani, A.S. Masur, S. Hegde, G.R. Vernekar, K. Siri Chandana, A survey on quantization methods for optimization of deep neural networks, in: International Conference on Automation, Computing and Renewable Systems, 2022, pp. 827–834.

[13] H. Li, A. Kadav, I. Durdanovic, H. Samet, H.P. Graf, Pruning filters for efficient ConvNets, in: International Conference on Learning Representations, 2017.

[14] X. Ding, X. Zhang, N. Ma, J. Han, G. Ding, J. Sun, Repvgg: Making vgg-style convnets great again, in: IEEE/CVF Conference on Computer Vision and Pattern Recognition, 2021, pp. 13733–13742.

[15] F. Yu, L. Cui, P. Wang, C. Han, R. Huang, X. Huang, EasiEdge: A novel global deep neural networks pruning method for efficient edge computing, IEEE Internet Things J. 8 (2021) 1259–1271.

[16] H. Liu, K. Simonyan, Y. Yang, DARTS: Differentiable architecture search, in: International Conference on Learning Representations, 2019.

[17] H. Wang, C. Ge, H. Chen, X. Sun, PreNAS: Preferred one-shot learning towards efficient neural architecture search, in: Proceedings of the 40th International Conference on Machine Learning, Vol. 202, 2023.

[18] W. Jiang, Y. Chen, S. Wen, L. Zheng, H. Jin, PDAS: Improving network pruning based on progressive differentiable architecture search for DNNs, Future Gener. Comput. Syst. 146 (2023) 98–113.

[19] Y. LeCun, J. Denker, S. Solla, Optimal brain damage, in: Advances in Neural Information Processing Systems, Vol. 2, 1989.

[20] D. Blalock, J.J. Gonzalez Ortiz, J. Frankle, J. Guttag, What is the state of neural network pruning? in: Conference on Machine Learning and Systems, Vol. 2, 2020, pp. 129–146.

[21] K. Wu, Y. Guo, C. Zhang, Compressing deep neural networks with sparse matrix factorization, IEEE Trans. Neural Netw. Learn. Syst. 31 (2020) 3828–3838.

[22] Y. Wang, D. Li, R. Sun, NTK-SAP: Improving neural network pruning by aligning training dynamics, in: International Conference on Learning Representations, 2023.

[23] J. Frankle, M. Carbin, The lottery ticket hypothesis: Finding sparse, trainable neural networks, in: International Conference on Learning Representations, 2019.

[24] S. Wiedemann, K.-R. Müller, W. Samek, Compact and computationally efficient representation of deep neural networks, IEEE Trans. Neural Netw. Learn. Syst. 31 (2020) 772–785.

[25] S. Han, X. Liu, H. Mao, J. Pu, A. Pedram, M.A. Horowitz, W.J. Dally, EIE: Efficient inference engine on compressed deep neural network, ACM SIGARCH Comput. Archit. News 44 (2016) 243–254.

[26] J. Frankle, G.K. Dziugaite, D. Roy, M. Carbin, Pruning neural networks at initialization: Why are we missing the mark? in: International Conference on Learning Representations, 2021.

[27] H. Tanaka, D. Kunin, D.L. Yamins, S. Ganguli, Pruning neural networks without any data by iteratively conserving synaptic flow, Adv. Neural Inf. Process. Syst. 33 (2020) 6377–6389.

[28] X. Ma, S. Lin, S. Ye, Z. He, L. Zhang, G. Yuan, S.H. Tan, Z. Li, D. Fan, X. Qian, X. Lin, K. Ma, Y. Wang, Non-structured DNN weight pruning–Is it beneficial in any platform? IEEE Trans. Neural Netw. Learn. Syst. (2021) 1–15.

[29] P. Molchanov, S. Tyree, T. Karras, T. Aila, J. Kautz, Pruning convolutional neural networks for resource efficient inference, in: International Conference on Learning Representations, 2017.

[30] M. Alizadeh, S.A. Tailor, L.M. Zintgraf, J. van Amersfoort, S. Farquhar, N.D. Lane, Y. Gal, Prospect pruning: Finding trainable weights at initialization using meta-gradients, in: International Conference on Learning Representations, 2022.

[31] T. Choudhary, V. Mishra, A. Goswami, J. Sarangapani, Inference-aware convolutional neural network pruning, Future Gener. Comput. Syst. 135 (2022) 44–56.

[32] Y. Han, G. Huang, S. Song, L. Yang, H. Wang, Y. Wang, Dynamic neural networks: A survey, IEEE Trans. Pattern Anal. Mach. Intell. (2021).

[33] X. Wang, F. Yu, Z.-Y. Dou, T. Darrell, J.E. Gonzalez, Skipnet: Learning dynamic routing in convolutional networks, in: European Conference on Computer Vision, 2018, pp. 409–424.

[34] Z. Xu, F. Yu, J. Xiong, X. Chen, Helios: Heterogeneity-aware federated learning with dynamically balanced collaboration, in: 58th ACM/IEEE Design Automation Conference, 2021, pp. 997–1002.

[35] J. He, G. Baig, L. Qiu, Real-time deep video analytics on mobile devices, in: 24th International Symposium on Theory, Algorithmic Foundations, and Protocol Design for Mobile Networks and Mobile Computing, 2021, pp. 81–90.

[36] J. Frankle, G.K. Dziugaite, D.M. Roy, M. Carbin, Linear mode connectivity and the lottery ticket hypothesis, in: International Conference on Machine Learning, 2020.

[37] S. Ioffe, C. Szegedy, Batch normalization: Accelerating deep network training by reducing internal covariate shift, in: International Conference on Machine Learning, 2015, pp. 448–456.

[38] P. Deutsch, DEFLATE Compressed Data Format Specification version 1.3, RFC 1951, RFC Editor, 1996, URL https://www.rfc-editor.org/rfc/rfc1951.txt.

[39] F. Romero, Q. Li, N.J. Yadwadkar, C. Kozyrakis, INFaaS: Automated model-less inference serving, in: USENIX Annual Technical Conference, 2021, pp. 397–411.

[40] T. Pragnesh, B.R. Mohan, Compression of convolution neural network using structured pruning, in: 7th IEEE International Conference for Convergence in Technology, 2022, pp. 1–5.

[41] M. Ganaie, M. Hu, A. Malik, M. Tanveer, P. Suganthan, Ensemble deep learning: A review, Eng. Appl. Artif. Intell. 115 (2022).

[42] A. Krizhevsky, G. Hinton, et al., Learning multiple layers of features from tiny images, 2009.

[43] Y. Le, X. Yang, Tiny imagenet visual recognition challenge, 2015.

[44] K. He, X. Zhang, S. Ren, J. Sun, Deep residual learning for image recognition, in: IEEE Conference on Computer Vision and Pattern Recognition, 2016, pp. 770–778.

[45] A. Dosovitskiy, L. Beyer, A. Kolesnikov, D. Weissenborn, X. Zhai, T. Unterthiner, M. Dehghani, M. Minderer, G. Heigold, S. Gelly, J. Uszkoreit, N. Houlsby, An image is worth 16x16 words: Transformers for image recognition at scale, in: International Conference on Learning Representations, 2021.

[46] P. Hill, A. Jain, M. Hill, B. Zamirai, C.-H. Hsu, M.A. Laurenzano, S. Mahlke, L. Tang, J. Mars, Deftnn: Addressing bottlenecks for DNN execution on GPUs via synapse vector elimination and near-compute data fission, in: 50th Annual IEEE/ACM International Symposium on Microarchitecture, 2017, pp. 786–799.

[47] A. Khetan, Z. Karnin, Prunenet: Channel pruning via global importance, 2020, arXiv preprint arXiv:2005.11282.

[48] Y. Jiang, S. Wang, V. Valls, B.J. Ko, W.-H. Lee, K.K. Leung, L. Tassiulas, Model pruning enables efficient federated learning on edge devices, IEEE Trans. Neural Netw. Learn. Syst. (2022).

[49] M. Shen, P. Molchanov, H. Yin, J.M. Alvarez, When to prune? A policy towards early structural pruning, in: IEEE/CVF Conference on Computer Vision and Pattern Recognition, 2022, pp. 12237–12246.

[50] Y. Cai, W. Hua, H. Chen, G.E. Suh, C. De Sa, Z. Zhang, Structured pruning is all you need for pruning CNNs at initialization, 2022, arXiv preprint arXiv:2203.02549.

Bailey J. Eccles received the M.Eng degree in Computer Science from Queen's University Belfast, UK in 2021. He is currently pursuing a Ph.D. degree in Computer Science at the University of St Andrews, UK. His major interests are in the areas of machine learning, edge computing, model compression and optimisation.

Philip Rodgers received the Ph.D. degree in Computer Science from the University of Strathclyde, UK in 2019. He is currently a Research Scientist at the Rakuten Mobile Innovation Studio, Japan. His research interests are in distributed algorithms for artificial intelligence.

Peter Kilpatrick received the Ph.D. degree in Computer Science in 1985, and the B.Sc. degree in Mathematics and Computer Science in 1981. He is currently a Reader in Computer Science at Queen's University Belfast. His interests include parallel programming models and cloud and edge computing.

Ivor Spence received the Ph.D. degree in Computer Science from Queen's University Belfast, UK, where he did research on code generation. He is currently a Reader in computer science at Queen's University Belfast where he leads the artificial intelligence (AI) research theme. His research is primarily on heterogeneous computing systems for AI.

Blesson Varghese received the Ph.D. degree in Computer Science from the University of Reading, UK on international scholarships. He is a Reader in Computer Science at the University of St Andrews, UK, and the Principal Investigator of the Edge Computing Hub. He is a previous Royal Society Short Industry Fellow. His interests include distributed systems that span the cloud–edge-device continuum and edge intelligence applications. More information is available from www.blessonv.com.