UNIVERSITY OF SOUTHAMPTON

Faculty of Engineering and Physical Science
School of Electronics and Computer Science

An Investigation into Weight Fixing Networks

by

Christopher Subia-Waud

Doctor of Philosophy Thesis

May 2025

Contents

List of Figures

List of Tables

Acknowledgements

Deepest thanks to all those who have supported me along the way, and to the pursuit of interestingness, for keeping both life and research ever so slightly off-kilter and delightfully unpredictable.

Chapter 1

Introduction

Deep learning models have a seemingly inexorable trajectory toward growth. Growth in applicability, performance, investment, and optimism. Unfortunately, one area of growth is lamentable - the ever-growing energy and storage costs required to train and make predictions. A considerable factor contributing to these high energy costs is the hardware they run on, which has historically been a function of the organisation of computation and the types of computation required.

Just as the nature versus nurture debate oscillates between the relative influence of innate abilities compared to environmental influences, the success of deep learning equally hinges on the dynamic interplay between algorithmic development and hardware advancements. Historically, this algorithm-hardware dance was skewed towards adapting algorithms to pre-existing hardware, much like shaping behaviors based on the environment in the nature-nurture analogy.

This environment of adapting algorithms to hardware has been punctuated by specific periods of high research investment into dedicated hardware, notably during the early '90s. The development of ANNA [Sackinger et al., 1992] and Intel's ETANN [IEEE and IEEE, 1992] are examples of these efforts, harmonising more with the aspect of 'nature' or innate abilities. However, those models were non-flexible, supporting only fixed architecture models, akin to fixed or inherited traits.

The Pivotal turn in deep learning was the innovative repurposing of Graphics Processing Unit (GPU) hardware. Initially developed to empower the video game industry, GPUs offered a flexible and powerful resource to advance and refine machine learning algorithms.

It is crucial to make the distinction between algorithm-hardware co-design and hardware re-purposing. The nascent success of deep learning systems was far more of a

function of the latter than the former. However, with the dominant view that further scaling with increased performance [Kaplan et al., 2020] coupled with the ever-greater reach of these models into the modern economy, there is a growing thrust toward finding bespoke hardware accelerators to minimise the energy consumption of these powerful models. However, the energy cost of these models is not just a function of the computations performed, but also the data movement required to support those computations. Data movement refers to the transfer of data between different levels of the memory hierarchy, such as between main memory and cache, or between different processing units, which can consume significant amounts of energy.

Whilst some algorithmic techniques have been useful in supporting this goal – one example being weight quantisation to INT8 which is supported out-the-box by TPUs [Jouppi et al., 2017a] – far too many have been concentrated on benchmarking metrics such as compute floating-point operations (FLOPs) and its closely related multiply and accumulate (MAC), without a clear cut direct translation into hardware not so clear-cut [Sze et al., 2017]. As we will discuss, this is largely because the data movements carry the dominant energy costs, which are not well captured with these popular metrics.

This was the starting point for this thesis — the assertion that the most resource-expensive component of neural network inference is borne by data movement, and how can we find algorithmic techniques which reduce this.

Centred around the consideration of the dominance of data-movement costs, the initial work of this thesis focuses on finding a set of algorithmic methods to reduce the number of *unique* weights along with the weight-space entropy of neural networks - which we will later show are key to maximising opportunity for data-movement reductions. The methods were developed to reduce the inference costs of models. They were tested on image-type classification problems; although in principle there is no reason that they can not be extended and applied to other problem domains (such as text prediction).

The path through to writing this thesis was not satisfactorily concluded upon developing such a method. Instead, we noticed a link between the method developed and Bayesian Neural Networks (BNNs) — a subfield in deep learning which looks to reformulate neural networks to be stochastic in order to gain a measure of network uncertainty.

With this link identified, we will later explore and use this link to provide further improvements in determining models with very few unique weight and low weight-space entropies. This link then further stimulated additional study into uncertainty estimates with large-scale model-dataset BNNs which hitherto were restricted to the toy-end of Machine Learning problems.

We ultimately hope with this thesis to explore, communicate and find the links between five core statements:

- Reductions in data movement confer significant benefits for AI hardware energy optimisation - Chapter 2.
- To minimise data movement, it is advantageous to reduce the number of unique weights and decrease weight-space entropy - Chapters 2 and 3.
- The flexibility in determining which weights can correspond to which unique values is magnified when we enhance the noise-tolerance of the network - Chapters 3 and 4.
- A BNN that maximises noise-tolerance is a boon for compression, can be trained on a wider set of model-dataset combinations - Chapters 4 and 5
- BNNs trained in this way are also well calibrated and can detect out-of-distribution samples - Chapter 6.

The thesis is structured as follows:

Chapter 2 discusses the hardware costs and opportunities in deep learning inference, highlighting the dominance of data movement costs.

Chapter 3 introduces Weight Fixing Networks (WFN), an algorithmic approach to reduce unique weights and weight-space entropy to enable more efficient hardware designs. The chapter concludes by identifying limitations in WFN's assumption about weight movement.

Chapter 4 proposes Probabilistic Weight Fixing Networks (PWFN), which reformulates WFN using Bayesian Neural Networks to address the limitations of WFN. PWFN achieves further reductions in weight-space entropy and unique parameters.

Chapter 5 builds upon PWFN and introduces Cluster-On-the-Fly PWFN (COF-PWFN), which integrates the clustering process into the training phase for a more adaptive and streamlined approach.

Chapter 6 explores the potential of PWFN and COF-PWFN for providing uncertainty estimations, investigating their calibration capabilities and ability to detect out-of-distribution samples.

Let us begin by looking where the energy-costs in hardware lie.

Chapter 2

Background

2.1 Costs and Opportunities

All operations in computation bear the cost of data movement and data manipulation, such as arithmetic operations. Computational cost in deep learning inference is measured by evaluating these two aspects. A single inference calculation involves fetching all the elements needed like weights, partial sums, and inputs, and then conducting several elementary operations – multiplications and summations – to map inputs to outputs.

As shown in Table 2.1, data movement and arithmetic operations carry different energy costs. Notably, the cost varies significantly between types of memory: Static Random Access Memory (SRAM) - a high-speed memory used for cache, and Dynamic Random Access Memory (DRAM) - typically used for data or program code needed by a computer processor to function. Furthermore, arithmetic operations like addition ($+$) and multiplication ($\times$) performed on different data types from 8-bit integers to 32-bit floating-point types, consume energy orders of magnitude cheaper than data movement between SRAM and DRAM [Kestor et al., 2013, Boroumand et al., 2018, Keckler et al., 2011, Pandiyan and Wu, 2014, Horowitz, 2014, Jouppi et al., 2021]. Transitioning data fetching from DRAM to SRAM leads to more significant energy saving than moving from 16-bit to 8-bit arithmetic operations.

The results presented in the table correspond to two manufacturing process technologies represented by their transistor sizes: 45 nanometers (nm) and 7 nm - a technological capability indicator where smaller size signifies advancement leading to better performance and energy efficiency.

Operation		Picojoules per Operation	
		$45nm$[Horowitz, 2014]	$7nm$[Jouppi et al., 2021]
$+$	Int8	0.03	0.007
	Int32	0.1	0.03
	BFloat16	-	0.11
	IEEE FP 16	0.4	0.16
	IEEE FP 32	0.9	0.38
$\times$	Int8	0.2	0.07
	Int32	3.1	1.48
	BFloat16	-	0.21
	IEEE FP 16	1.1	0.34
	IEEE FP 32	3.7	1.31
SRAM	8KB	10	7.5
	32KB	20	8.5
	1MB	100	14
DRAM	DDR3/4	1300	1300
	HBM2	-	$250 - 450$
	GDDR6	-	$350 - 480$

TABLE 2.1: Energy costs of elementary computation operations in picojoules. The operations are performed across different data types and storage means (SRAM and DRAM), with the table comparing results from two transistor sizes used in the production of microprocessors: 45 nm and 7 nm.

Fortunately, there is something special about the types of computational requests we make for deep learning systems that allow accelerator designs potential ways to reduce these dominant costs. Namely, the fetched data used in inference calculations are seldom used in the computation only once, giving us re-use opportunities that can potentially ameliorate the total energy costs through a reduction in the number of memory reads. This, coupled with matrix sparsity and hierarchical computation, offers multiple avenues for a reduced energy overhead. The convolution network is an excellent example of the opportunities available in reducing energy costs; let us briefly dive into their history to see how such opportunities evolved.

The key take away here is that data reads are orders of magnitude more costly than arithmetic operations. To reduce the energy costs, a focus must be on the re-use of data. Read once, use many times.

2.1.1 The Convolution Layer

The groundbreaking work of Nobel laureates Hubel and Wiesel on understanding the functional architecture of the visual cortex [Hubel and Wiesel, 1962] not only revolutionised our understanding of biological vision but also laid the foundation for the development of energy-efficient deep learning algorithms. Through their innovative

studies on the structure of neurones within a cat's visual cortex, they discovered a spatially hierarchical organisation of neuronal interactions.

Stimulation of the cortex was done by visually presenting simple patterns, such as straight lines or edges, to the cat. It was observed that lateral movement of the electrode across the cortex changed the type of feature detectors encountered—from vertical line detectors to 'L'-shape detectors, for instance. This suggests that neurones in close geographic proximity specialise in detecting different types of visual features (i.e., an edge '—' vs an 'L'). On the other hand, vertical movement of the electrode along the depth of the cortex fixed the type of detected feature but changed the receptive field, or specific area of the visual field to which neurones responded.

Moreover, Hubel and Wiesel noted that nearby neurones had overlapping receptive fields, meaning they responded to common portions of the visual field. This shared spatial distribution, along with the hierarchical organisation of feature detection, directly influenced the design of convolutional neural networks (CNNs) where the weight-sharing and hierarchical structure of CNNs bear a close resemblance to the overlapping receptive fields and hierarchical organisation observed in the visual cortex.

The seminal work of [Fukushima and Miyake, 1982] aimed to replicate aspects of this hierarchical organisation in silicon with the Neocognitron. This early model incorporated key architectural design choices that have carried through to modern deep learning systems and are central to the pursuit of energy-efficient hardware: a layered hierarchy and weight/output sparsity.

In CNNs, the layered hierarchy and filter re-use inherently reduce data movement by enabling the sharing of weights across multiple spatial locations and inputs. This weight-sharing, introduced in [Rumelhart et al., 1986, LeCun et al., 1989], enforces translation equivariance properties. Translation equivariance means that if an input image is shifted, the output of the network will be shifted in the same way, allowing the network to recognize features regardless of their position in the image. This property is a key area where research has focused on minimising energy costs. The energy efficiency of these operations can be further improved by reducing the number of unique weights and decreasing the weight-space entropy, which is a core objective of this thesis.

In addition to weight-sharing, the sparsity of weights and outputs in CNNs, inspired by the sparse activation patterns observed in the visual cortex, presents further opportunities for energy optimisation. By exploiting this sparsity, we can minimise the number of computations and data movements required, thus reducing overall energy consumption.

In the following sections, we will explore algorithmic techniques that push further these architectural features of CNNs–weight-sharing and sparsity—to reduce inference costs, but first let's more formally define what exactly a deep neural network (DNN) is.

2.1.2 The Three Key Properties

There are three properties of modern deep learning architectures, two of which were already present in the early work of [Fukushima and Miyake, 1982], that can be leveraged by hardware to reduce computational expense.

- A Layered Input-output Neuronal Hierarchy: Deep learning architectures, such as CNNs, are organised into a hierarchical structure of layers. Each layer processes the output from the previous layer and passes its output to the next layer. This hierarchical organisation allows for the extraction of increasingly complex features as the input progresses through the network. By designing hardware that efficiently handles this layered structure, computational expense can be reduced.
- Weight/Output Sparsity: In deep learning models, many of the learned weights and activations (outputs) are often zero or near-zero. This sparsity can be exploited by hardware accelerators to reduce computation and memory access. By avoiding calculations and data movements involving zero values, the overall computational expense can be minimised. Techniques such as pruning and sparse matrix operations can be used to take advantage of this sparsity.
- Weight/Input Pixel Re-use: In CNNs, the same set of weights (filters) are applied across different spatial locations of the input. This weight re-use, also known as weight sharing, allows for a significant reduction in the number of unique weights that need to be stored and accessed. Similarly, input pixels can be re-used across multiple convolution operations. Hardware accelerators can leverage this re-use by efficiently caching and reusing weights and input pixels, reducing memory access and computational expense.

2.1.3 Layered Input-Output Neuronal Hierarchy

A DNN, at its most universal definition, is a stack of L layers, each of which acts on inputs to layer l, denoted as $\mathbf{h}^{l-1}$, and applies a function $F^l_{w^l}$ at a particular layer l. The function $F^l_{w^l}$ is parameterised by weights w^l. The input to the network is denoted as $\mathbf{h}^0 = \mathbf{x}$, and the output of the network is $\mathbf{h}^L = \mathbf{y}$.

In the typical case, the output of a layer l is fed into as input to the next layer in a cascading fashion:

$$\mathbf{y} = F_\theta(\mathbf{x}) = F^L_{w^L} \circ F^{L-1}_{w^{L-1}} \circ \cdots \circ F^1_{w^1} \circ F^0_{w^0}(\mathbf{x})$$

where $\mathbf{h}^{l+1} = F^l_{w^l}(\mathbf{h}^l)$, $h^0 = x$, $\mathbf{h}^{L+1} = y$h is the output (or hidden state) of layer l, and $\theta = (w^L, w^{L-1}, \ldots, w^0)$ is the collection of all weights in the network.

The types of inputs $F^l_{w^l}$ expects and how it acts on such inputs carries a great deal of variety and freedom, but this layered structure has consequences for hardware use. Each layer $F^l_{w^l}$ can be treated as a self-contained input-to-output function, meaning that weights w^l along with the inputs $\mathbf{h}^{l-1}$ are all required to be loaded from memory.

This independent computation allows for layer-by-layer compression techniques, such as quantisation, to be applied more effectively than compressing the entire network at once. By treating each layer as a separate entity, the compression algorithm can be tailored to the specific characteristics and requirements of each layer, potentially leading to better compression ratios and less loss of information. In contrast, when compressing the entire network at once, the compression algorithm must find a one-size-fits-all solution, which may not be optimal for all layers. Furthermore, layer-by-layer compression enables the possibility of using different compression techniques or parameters for each layer, which can further improve the overall compression efficiency.

2.1.4 Weight Sparsity

In modern deep neural networks, weight sparsity is often induced through techniques such as pruning, weight-decay regularisation, and quantisation. Pruning involves removing weights that are close to zero or have minimal impact on the network's performance, effectively setting them to zero. Weight-decay regularisation adds a penalty term to the loss function, encouraging the network to learn smaller weights, which can lead to increased sparsity. Quantisation, on the other hand, reduces the precision of weights, which can result in more weights being exactly zero.

Another form of sparsity, activation sparsity, arises from the use of activation functions such as the Rectified Linear Unit (ReLU). ReLU sets all negative input values to zero, introducing sparsity in the activations.

The presence of sparse weights and activations has significant benefits for hardware design. In computations involving addition, zero-valued weights and activations have no effect on the result. Similarly, in multiplications, zero elements always map back to

zero. By exploiting these properties, hardware systems can avoid performing unnecessary arithmetic operations, leading to energy savings. If the prevalence of zero weights and activations can be increased through the aforementioned techniques, and hardware systems are designed to identify and skip these operations, substantial efficiency gains can be achieved.

2.1.5 Re-use Opportunities

Designing efficient deep learning accelerators requires focusing on data reuse and locality to minimise data movement costs and improve computational efficiency. These considerations are closely related to the layered structure of deep neural networks and the independent computation within each layer, as discussed in the previous section on weight sparsity.

Deep learning computations offer multiple opportunities to reuse parameters, directly addressing the key considerations of data reuse and locality. By reading each parameter once and applying it to every instance where it is needed, the overall computational burden can be significantly reduced. Common operations and layers in deep learning architectures, provide several opportunities for parameter reuse. Let's take a look at the convolution layer as one such example:

- **Convolutional Filter Reuse**: In convolutional layers, each filter is applied to multiple spatial locations across the input feature map. This means the filter parameters can be read once and reused for all the spatial positions, reducing the need to repeatedly load the same filter from memory. Exploiting this reuse opportunity can minimize data movement and improve computational efficiency.
- **Input Reuse**: Each spatial location within the input feature map contributes to multiple output feature maps. In other words, a single input value is used in the computation of several output values, as it is multiplied by different filter parameters. Keeping the input in local memory or registers can avoid redundant memory accesses and improve computational efficiency. This directly relates to the importance of data locality in accelerator designs.
- **Batch Reuse**: When processing data in batches, the same filters are applied to multiple input examples within the batch. This presents an opportunity to reuse the filter parameters across the batch, further reducing memory access requirements and enabling parallel processing of multiple examples. Batch reuse is particularly relevant to the concept of parallelism in accelerator designs, as it allows for efficient utilization of computational resources.

Next, in following sections will look at how these reuse opportunities can be exploited in the design of hardware architectures and algorithms for deep learning acceleration.

2.2 Hardware for Deep Learning Inference

Hardware used for DNN inference can be divided into two categories: general and specialist. General hardware systems, whilst not explicitly designed for DNN inference calculations, can perform these tasks due to the simplicity of the underlying operations, such as memory reads and matrix multiplications. On the other hand, specialist DNN inference machines, or accelerators, are engineered specifically to carry out the operations required for inference and to reduce any overhead or bottlenecks. These specialised accelerators have been introduced to address the inefficiencies present in systems that support more general computation.

Some of these inefficiencies include the high area costs incurred in supporting large numbers of operators, the high energy and latency costs due to highly connected, multi-layered memory hierarchies typically implemented in general computing, and the smaller on-chip RAM sizes often requiring multiple high-energy cost off-chip accesses. These inefficiencies can be particularly pronounced when dealing with the large number of weights and parameters present in modern deep neural networks.

Whilst significant progress in AI has been achieved using general hardware systems, the slowing trend of Moore's Law [Moore, 1998] and the inherent inefficiencies of a one-size-fits-all approach highlight the potential of domain-specific hardware components for further advancement [Hennessy, 2018, Hameed et al., 2010]. By focusing on the specific requirements of DNN inference, specialist architectures can be designed to optimise computational efficiency and minimise data movement, leading to improved performance and energy efficiency.

One key aspect of this optimisation is the handling of weights and parameters in the network. As discussed in previous sections, deep neural networks often have a large number of weights, which can lead to increased memory requirements and data movement costs. By exploring techniques such as weight sparsity, quantisation, and efficient weight storage, specialist architectures can significantly reduce the memory footprint and computational burden associated with these weights.

Furthermore, the reuse opportunities present in common DNN operations, such as convolution, can be exploited by specialist architectures to minimise data movement and improve computational efficiency. By carefully designing memory hierarchies and dataflow patterns that take advantage of these reuse opportunities, specialist

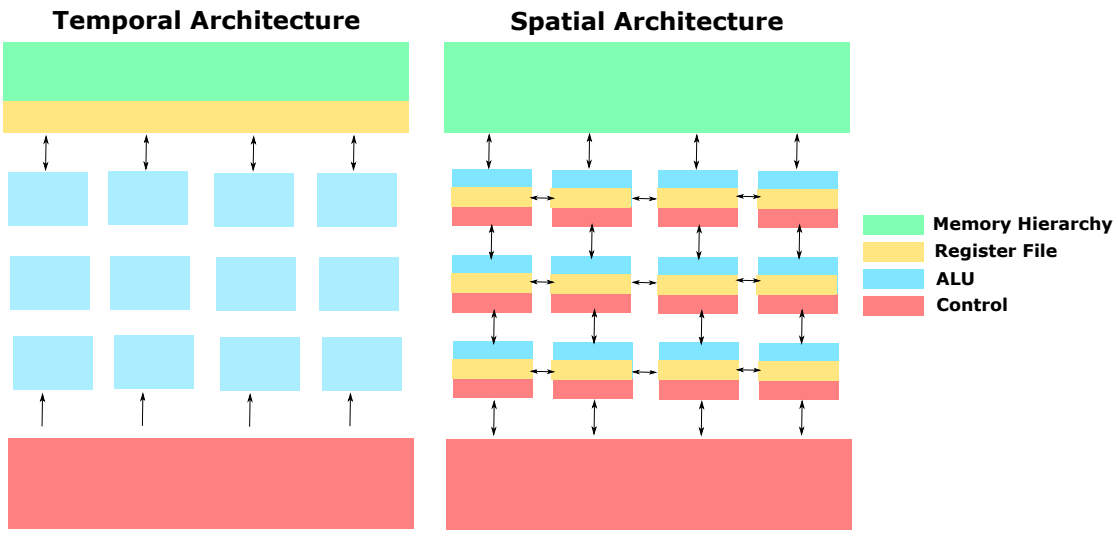


FIGURE 2.1: Temporal architectures – such as GPU's – have processing element ALUs that carry out computation in parallel and interact directly with control and memory, but not each other. These architectures use SIMD/SIMT to increase processing efficiency. Spatial architectures – DNN accelerators – have processing element which have in addition, programmable register files and can communicate with each other and use dataflowprocessing schemes to reduce data movement.

architectures can achieve significant performance gains compared to general hardware systems.

In the following sections, we will delve deeper into the architectural details that distinguish specialist architectures from general systems and explore how these design choices are motivated by the specific requirements of DNN inference. By understanding these differences and their implications for weight handling and data reuse, we can gain insights into the development of efficient deep learning accelerators and the potential for further optimisation through techniques such as weight space entropy reduction.

2.2.1 Temporal Architecture

The two staples of modern computing, central processing units (CPUs) and graphics processing units (GPUs), both follow a temporal architecture. In this type of architecture, a single centralised control unit distributes work to multiple arithmetic logic units (ALUs). These ALUs have access to a shared memory but do not typically have their own programmable memory store, meaning they need to rely on access to shared memory pools. This reliance on shared memory can cause bottlenecks due to the growing gap between processor and memory performance, known as the memory wall [McKee and Wisniewski, 2011].

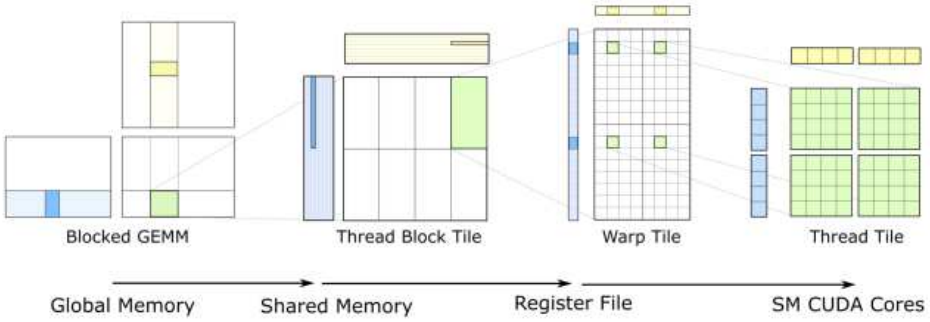


FIGURE 2.2: Tiling approaches allow the breaking of large matrix multiplications prevalent in DNN inference into many reusable blocks which can be operated on in parallel and reused – saving large memory reads and writes.

To improve computation speed through parallelism, techniques such as Single Instruction Multiple Data (SIMD) and Single Instruction Multiple Threads (SIMT) are employed. SIMD, commonly used in CPUs, allows a single instruction to be applied simultaneously to multiple data elements, enabling parallel processing of identical operations on different data points. On the other hand, SIMT, often used in GPUs, extends the SIMD concept by allowing multiple threads to execute the same instruction simultaneously on different data elements, providing an additional level of parallelism.

However, the ALUs within temporal architectures are not able to communicate with each other directly and must rely on the centralised memory pool to exchange data and intermediate results. This communication bottleneck can limit the efficiency of parallel processing, especially when dealing with large amounts of data or complex operations.

In contrast, spatial architectures, which will be discussed in more detail later, allow for direct communication between processing elements, reducing the reliance on shared memory and potentially alleviating the communication bottleneck. This direct communication can be particularly beneficial for deep learning inference, where the exchange of intermediate results between layers is a common operation.

The limitations of temporal architectures in terms of memory access and communication between ALUs highlight the need for specialised hardware that can address these bottlenecks. By designing architectures that are tailored to the specific requirements of deep learning inference, such as efficient memory access patterns and direct communication between processing elements, we can potentially achieve significant improvements in performance and energy efficiency.

2.2.1.1 Optimisation Strategies within Temporal Architectures

Temporal architectures and optimisation libraries – such as OpenBLAS, cuDNN etc – use strategies such as tiling to break up the matrix multiplications into reusable and parallelisable subblocks. These subblocks can themselves be hierarchical to match the memory hierarchy of the device as shown in Figure 2.2.

2.2.2 Spatial Architectures: A New Paradigm for DNN Inference

Spatial architectures offer a new approach to DNN inference, with two key features that distinguish them from temporal architectures. Firstly, they enable direct communication between processing elements (PEs), the individual units responsible for computation. Secondly, they provide programmable memories within each PE, allowing for fine-grained control over dataflow. These features provide access to close-to-compute memory and the ability to move data in unconventional ways, making spatial architectures well-suited for DNN inference, where reuse opportunities are abundant and the order of computation can be adjusted to meet specific requirements.

The programmable memories within each PE enable dataflow optimisation, allowing for the fine-tuning of data movement and computation order to suit the specific needs of the DNN inference task. This flexibility in dataflow mapping is a significant advantage, enabling the exploitation of reuse opportunities and minimisation of data movement costs.

Selecting the appropriate dataflow mapping is a critical aspect of designing spatial architectures for DNN inference. Three popular mappings – weight stationary, output stationary, and input stationary – each offer unique advantages (Figure 2.3). By investigating these mappings and understanding their implications for data movement and computation, we can design efficient spatial architectures that improve DNN inference performance.

The motivation behind spatial architectures and optimal dataflow mappings stems from the need to address the limitations of general-purpose hardware and temporal architectures in handling the large number of weights and complex dataflow patterns found in modern DNNs. By developing specialised hardware tailored to the specific requirements of DNN inference, significant improvements in computational efficiency, memory footprint, and energy consumption can be achieved.

Let's jump into a few definitions for some of the popular dataflow mappings with a mind that later we will look to develop our algorithmic approach to fit within one or more of these paradigms.

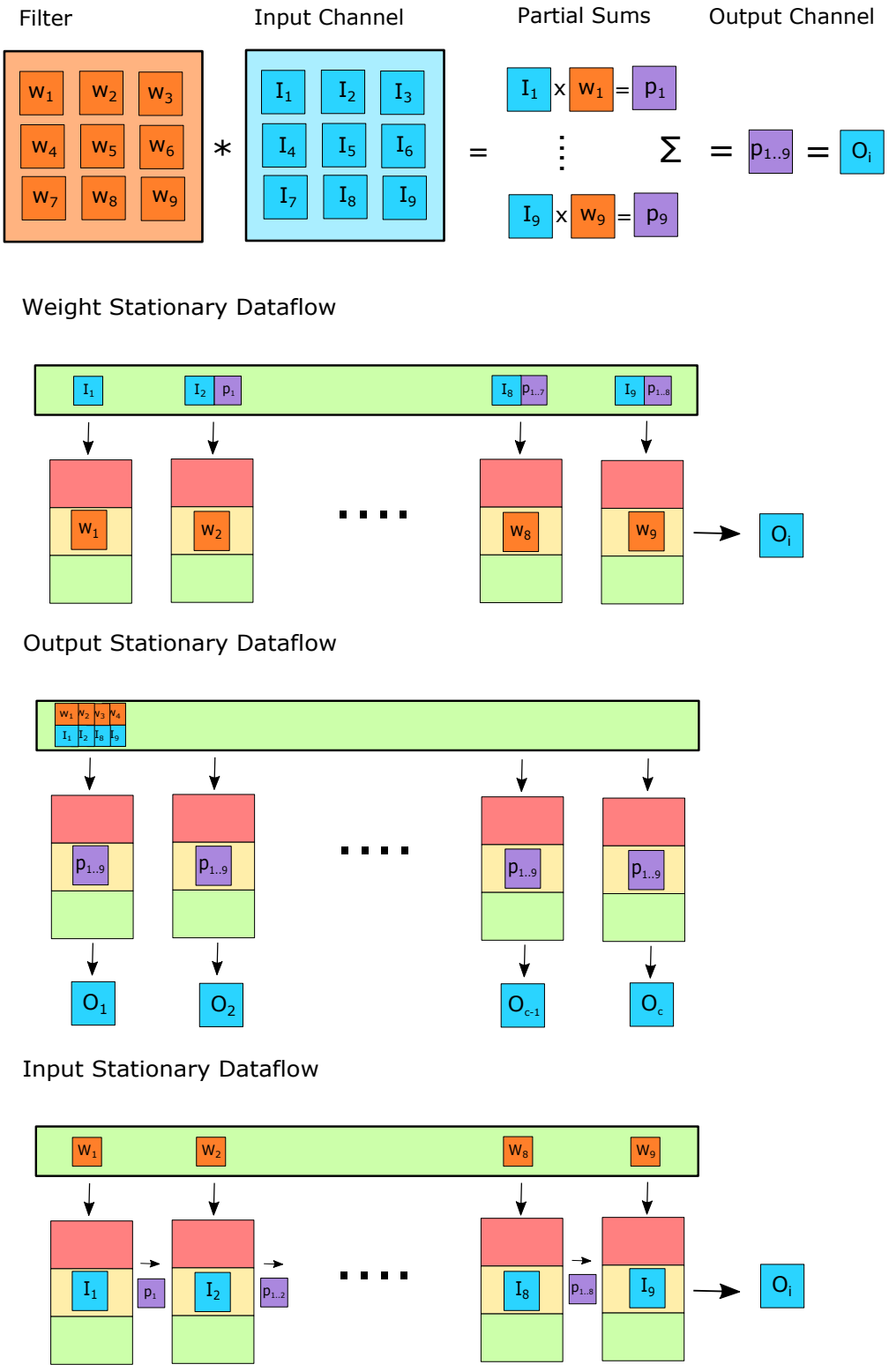


FIGURE 2.3: Dataflow mappings within accelerator designs prioritise the reuse of varying components of DNN computation. Weight stationery: maintains the weight values in PE memory and feeds in input activations and previous partial sums. Output stationery: maintains the partial sum accumulation in PE memory with inputs and weights fed in, meaning each PE outputs a single output activation channel – saving on the writing and re-reading of partial sums. Input stationery: keeps input activations within PE registry files feeding in weights and accumulating partial sums spatially between processing elements.

Weight Stationary Dataflow

Weight stationary dataflow optimises the local memory on PEs to hold the filter weights. The idea is to read each weight value only once and pass all input activations that use the weight in computation. This amounts to an off-chip memory-read saving of approximately the number of input channels to a filter weight for convolution. An early weight-stationary accelerator is DaDianNao [Chen et al., 2014] created a fully distributed system where there is no main memory shared by all PEs. Instead, each PE has access to a set of eDRAM banks and handles a subset of computation in a forward pass that uses these weights. Tensor processing units, a popular accelerator, also use weight stationary dataflow [Jouppi et al., 2017b] using a systolic array to orchestrate the input activations and gather the partial-sums as outputs to be passed in as inputs to other PEs, avoiding the other partial-sum read and writes. Other works that use weight-stationary dataflow include [Cavigelli et al., 2015, NVIDIA, 2018].

Output Stationary Dataflow

Output stationary dataflow minimises the energy consumption of the partial sum reading and writing for each output channel. Rather than having to write back to a buffer the intermediate computation for an output channel, we maintain partial sums on the PE registry file, and any weights and input activations involved in a single output channel are piped into this PE. The result is that each PE will output a single output channel result. This will approximately reduce the number of read-writes by the filter size used to compute each output channel. We can make further energy savings if we orchestrate the weights and/or inputs spatially such that neighbouring PEs use the same weight/input values, as was achieved in [Moons and Verhelst, 2016b, Du et al., 2015].

Input Stationary Dataflow

Following the same principle as the previous dataflows, input stationary dataflow maintains input activations in the registry files of the PEs and distributes the weights to the PEs that require each input. Partial sums are accumulated through communication between PEs without the need for writing back to off-chip memory. Each input held on a PE can therefore contribute to the computation of a volume of output activations determined by the filter dimensions (width and height) and the number of output channels. One architectural implementation that demonstrates this approach is SCNN [Parashar et al., 2017]. In this design, each PE maintains a bank of four input activations

and processes a portion of the output activation volume within a given cycle. The partial sums are then accumulated spatially within the four-input block.

Hybrid Dataflows

While weight, output, and input stationary dataflows cover the main ideas present in DNN accelerator designs, some researchers have explored the benefits of combining multiple dataflows through hybrid approaches [Parashar et al., 2017, Kwon et al., 2018, Chen et al., 2016, 2019]. One such example is the row stationary dataflow proposed by [Chen et al., 2016], which aims to maximise the reuse of weights, inputs, and partial sums for each filter row. In this approach, weights and inputs of a row are maintained within the PEs. Subsets of the row required for computation, as the filter moves across the row, can be accessed efficiently, and partial sums are collected for the row. Once a row is complete, the partial sums are combined with other rows within a filter's range to produce the final output.

2.2.3 Compressed Representations

Optimising the distribution and computational reuse of weights, inputs, and outputs across PEs is one way accelerators mitigate data movement costs; compression is another. Data compression is an established field with many ideas to borrow from, such as Shannon's information theory [Shannon, 1948b], Huffman coding [Huffman, 1952], and Lempel-Ziv-Welch (LZW) compression [Welch, 1984]. Rather than representing data in its full bit-width, compression schemes look for repeated structures to represent values or blocks with lower bit-width costs than the full precision values. In these schemes, values are stored using encoded indexing instead of directly storing the values themselves. These index schemes then point to the values' addresses in physical storage. For compression to be beneficial, the combined cost of indexing and value retrieval must be less than the traditional weight storage and read methods. This is particularly relevant to our work, as we aim to reduce the memory footprint and data movement costs associated with the weights in deep neural networks.

However, we are restricted to methods that can decompose subsets of the encoded data as needed for computation. If, as is the case for LZW, we can only decode the compression scheme in its entirety, then we would lose any benefit of compression at inference time.

Compressed representations that have been successfully used in accelerator designs include run-length encoding (RLE) [Chen et al., 2016], where integer values encode contiguously occurring zeros. For example, consider a row in a large sparse kernel with

values 0, 0, 9, 0, 0, 0, 0, 3; a simplified RLE representation might be z29z43, which is smaller than the original representation. However, this naive approach introduces ambiguity, as z29z43 could also represent 00043. In practice, RLE implementations for neural networks typically use separate, unambiguous encodings for zero runs and non-zero values, often with fixed-width formats or bit-level encodings. One problem with RLE, common to most compression schemes, is that the representational overhead may be more significant than the savings made. The effectiveness of RLE depends on the amount of sparsity in the matrix. Each contiguous set of zeros requires an integer to represent their number and an indicator that the zero blocks have started; if the matrix is not very sparse, or sparse but with the zero values interleaved without contiguous blocks, then the cost of RLE will be higher than the traditional dense representation. Other sparsity-optimised compression schemes used in accelerators include compressed sparse row and column formats [Sato and Tinney, 1963, Zhang et al., 2016].

While sparsity-leveraging compression schemes are popular, they seldom reach the limits of compression determined by the entropy of the weight distribution, which we would prefer. Ideally, the compression scheme used should be close to the underlying entropy of the weight distribution we wish to represent. An encoding scheme that has both properties, per-value decoding and close-to-entropy compression rates, is Huffman coding. This is particularly relevant to our work, as we aim to exploit the low-entropy weight distributions in deep neural networks to achieve high compression ratios while maintaining efficient decoding capabilities.

2.3 Hardware Realisation of Weight Compression

Transforming low-entropy weight distributions and reduced unique parameter values into efficient hardware requires specialised circuit designs that optimise both computation and memory access. When networks contain few unique weights, hardware accelerators can effectively exploit this property to minimise data movement costs.

Weight compression techniques can be realised in hardware through codebook-based architectures. Instead of storing full-precision weights, these systems maintain a small lookup table (LUT) of unique weight values in high-speed SRAM close to processing elements, while only indices to this table are stored in the main memory. This approach significantly reduces memory bandwidth requirements and enables more efficient data movement.

The UNPU accelerator [Lee et al., 2018] demonstrated this concept by supporting variable weight bit precision from 1 to 16 bits through lookup table-based bit-serial

processing elements. This architecture achieved energy reductions of 23.1%, 27.2%, 41%, and 53.6% for 16-, 8-, 4-, and 1-bit weight precision respectively, compared to conventional fixed-point multiply-and-accumulate arrays.

For networks containing primarily power-of-two weights, multiplications can be implemented as bit-shift operations, substantially reducing computational complexity. Modern accelerator designs can detect these special cases and route calculations through optimised datapaths.

Another approach to hardware acceleration is bit-level composability, as demonstrated by BitFusion [Sharma et al., 2018]. Rather than using fixed bitwidth processing elements, BitFusion employs an array of bit-level units that can dynamically fuse to match the bitwidth requirements of individual DNN layers. This flexibility allows for minimising computation and communication at the finest granularity possible without accuracy loss, achieving substantial performance and energy improvements over fixed-precision accelerators.

The EIE accelerator [Han et al., 2016a]. demonstrated specific hardware support for compressed neural networks, including Huffman-coded weights. Their implementation showed that the energy savings from reduced memory access substantially outweighed the decompression overhead, achieving a 3× reduction in total energy compared to conventional architectures when processing compressed networks.

These implementations underscore the critical importance of algorithm-hardware co-design—compression schemes must be developed with an understanding of hardware characteristics to maximise efficiency gains.

2.3.0.1 Huffman Coding

In this thesis, we dedicate two pieces of work (Chapters 3 and 4) to finding networks that minimise the Huffman encoded representation of a network, and so an expanded treatment of the subject is necessary.

In 1951, an MIT professor, Robert Fano (the joint inventor of Shannon-Fano coding), set an eager graduate class a coding challenge that, should they successfully solve, would exempt them from the upcoming final exam. A young David Huffman took up the challenge and spent months wrestling with iterations of ideas, not knowing that the problem was so tricky that Fano himself had no solution. Just days before the final exam, Huffman gave up the chase and accepted defeat, throwing his latest attempt into the bin and opening up his textbooks to prepare for the final exam. The following day, Huffman awoke with the realisation of a solution, one with only a slight divergence

from the idea that lay at the bottom of his waste paper basket. This aha moment led to the paper "A method for the construction of minimum-redundancy codes" [Huffman, 1952], which revolutionised the field and became the compression algorithm of choice for many applications. Huffman did not need to sit his final exam [Stix, 1991].

The algorithm that Huffman invented, like most good ideas, is a simple one. Starting with the string S we wish to encode, we represent each of the N unique symbols in the string as $A(S) = s_1, \cdots, s_n$, where $|S| = N$ and $|A(S)| = n \leq N$. Each of these unique symbols $s_i \in A(S)$ are initialised as $leaf$ nodes v_i in a node-set V. Each leaf node v_i additionally contains a weighting $\#(s_i)$, which represents the number of times the unique value is present in the string S. Let $V = v_1, \cdots, v_n$ represent the set of nodes, where $v_i = (s_i, \#(s_i))$ is a tuple containing the unique symbol and its corresponding frequency.

We next greedily select the two nodes v_i, v_j from V with the lowest weighting such that $i = \arg\min_k \#(s_k)$ and $j = \arg\min_l \#(s_l)$, where $k, l \in 1, \cdots, n$ and $k \neq i, l \neq j$. We remove these nodes from the set of possible nodes and attach in their place a combined parent node $v_{(i,j)} = ([s_i, s_j], \#(s_{(i,j)}))$, where the weighting of the parent node is the summation of the weights of the two child nodes selected, i.e., $\#(s_{(i,j)}) = \#(s_i) + \#(s_j)$. The new node set becomes:

$$V = (V \setminus \{v_i, v_j\}) \cup \{v_{(i,j)}\}$$

where $V \setminus \{v_i, v_j\}$ represents the set V with nodes v_i and v_j removed. If we repeat the process $n - 1$ times, we end with just one node that is parent to all other nodes $v_{(1,\cdots,n)}$. We measure the code word length for a particular symbol as the number of times the symbol was involved in combinations, and its bit string can be formed by traversing the tree created in the process, where each left or right path appends a 0 or 1 to the symbol representation, respectively.

Let us run through a concrete example, which we borrow heavily from [Moffat, 2019]. Imagine we have a very simple 1-layer neural network (essentially a linear regression setup) with the following 21 weights post-quantisation:

$$W = [0.3, 0, 0, 0.1, 0, 0, 0.5, 0, 0, 0.1, 0.1, 0, 0.1, 0.2, 0.2, 0.1, 0.4, 0, 0, 0, 0.1]$$

where $A(W) = \{0, 0.1, 0.2, 0.3, 0.4, 0.5\}$ and
$V = \{(0, 10), (0.1, 6), (0.2, 2), (0.3, 1), (0.4, 1), (0.5, 1)\}$.

In each iteration, we select the two nodes with the smallest frequencies. We use higher-numbered symbols as tie-breakers. Let's denote combined nodes with square brackets. The iterations proceed as follows:

$$
\begin{aligned}
&\text{Iteration 1:}\{(0,10),(0.1,6),(0.2,2),(0.3,1),([0.4,0.5],2)\}\\
&\text{Iteration 2:}\{(0,10),(0.1,6),([0.2,0.3],3),([0.4,0.5],2)\}\\
&\text{Iteration 3:}\{(0,10),(0.1,6),([[0.2,0.3],[0.4,0.5]],5)\}\\
&\text{Iteration 4:}\{(0,10),([0.1,[[0.2,0.3],[0.4,0.5]]],11)\}\\
&\text{Iteration 5:}\{([0,[0.1,[[0.2,0.3],[0.4,0.5]]]],21)\}
\end{aligned}
$$

We can represent our result in a tree as shown in Figure **??**, where a 0-1 traversal indicates the decisions made and the bit appended to the code for each leaf symbol.

Huffman coding has several properties that make it interesting for compressing neural networks. First, it is a *minimal-redundancy* code, meaning that the average number of coding digits per message is minimised. More importantly, Huffman encoding gets us very close to the underlying entropy. Defining entropy in the usual way, $\mathcal{H}(W) = -\sum_{w \in A(W)} p(w) \log_2 p(w)$, we can define an encoding cost as the total number of bits needed to encode our original weight set W as $\mathcal{H}'(C, W) = \sum_{w \in A(W)} \#(w) \cdot \mathcal{L}(w)$. The ϵ-difference between our encoding scheme C and the underlying entropy $\mathcal{H}$ is the quantity we would like to minimise: $\epsilon = \mathcal{H}'(C, W) - \mathcal{H}(W)$. If we define the most frequent weight in W as $w_{\max}$, we can use the bound presented in the work [Gallager, 1978], $0 \leq \epsilon < p(w_{\max}) + 0.086$, to gain an intuition of how close Huffman coding is to the underlying entropy of a message. Here we can see that the bound is limited by the probability of the most frequent weight in the code. For neural networks, this could present a problem since it is often empirically observed that the distribution of weights follows a Gaussian distribution or one similar but with longer tails [Yuhang Li, Xin Dong, 2020] - the Laplace distribution, for example. The Laplace distribution, defined by its probability density function $f(x|\mu, b) = \frac{1}{2b} \exp(-\frac{|x-\mu|}{b})$ where μ is the location parameter and b is the scale parameter. The Gaussian has contributed to the developments of informative priors for Bayesian Neural Networks [Blundell et al., 2015] and has been utilised by pruning techniques to fix as much as 90% of the weights to zero values [Lin et al., 2020b]. If the most frequent weight has a probability of 0.9, then the worst-case scenario is that Huffman coding could add up to 0.986 bits per weight of overhead. Fortunately, as we will discuss, this need not be the case, but it is still a result worth keeping in mind as we go forward.

$$\epsilon(C, S) = \mathcal{H}'(C, S) - \mathcal{H}(S) \tag{2.1}$$

The bound on the ϵ-difference between Huffman coding and the underlying entropy is given by:

$$\epsilon(C,S) \leq \begin{cases} \frac{\#(s_{max})}{n} + 0.086 & \text{when } \frac{\#(s_{max})}{n} < \frac{1}{2} \\ \text{when } \frac{\#(s_{max})}{n} \geq \frac{1}{2} & \frac{\#(s_{max})}{n} \end{cases} \tag{2.2}$$

Let us wrap up our initial discussion of Huffman coding with a list of properties that make them useful when thinking about an encoding scheme for Neural Networks:

- It is a prefix code, hence it can be decoded one symbol at a time. This is helpful for DNN inference where individual weights used in computation are usually paired with elements of the input signal. Hence, we could decode a single weight, use it, and then move onto the next part of computation rather than having to decode the network in its entirety.
- We have theoretic bounds on how close the encoding is to the underlying entropy [Gallager, 1978], so we can look to adapt our distribution shaping methods with this in mind.
- The decoding can be done on the fly using an additional buffer memory of size $\mathcal{L}_{max}$ bits [Moffat, 2019].
- Finally, Huffman encoding has been successfully implemented and used in accelerator designs with an 8x energy saving achieved when using this form of encoded weight-sharing [Mao and Dally, 2016a, Han et al., 2016a]. This successful implementation is an indication of the potential benefits of low-entropy networks for hardware efficiency and serves as an inspiration for our work in finding improved weight distributions that can exploit low-entropy encodings. By exploring techniques to shape the weight distribution and reduce the entropy of the network, we aim to further enhance the effectiveness of Huffman coding and other entropy-based compression schemes, ultimately leading to more efficient deep learning accelerators.

2.4 Algorithmic Approaches

Let us now start to move towards the algorithmic side of DNN acceleration. This section differs from the previous section in that these methods are typically developed agnostically, with no particular hardware configuration in mind. Instead, the network is altered or designed with the general principles of downstream energy efficiency. We note, however, often a particular type of hardware system or module is required to realise the energy savings of techniques employed using algorithmic approaches.

We break down our dive into the algorithmic approaches to DNN inference energy saving as pruning, quantisation, distillation and architectural choice and search. These four categories benefit (at least for the most part) from being distinct so that a practitioner can use a single technique for a project or combine two or more for further gains.
Before we begin, let us prime ourselves with a reminder of the common principles explored in the previous hardware section:

- **Data Movement Domination**: The cost of moving data from off-chip DRAM far surpasses the costs involved in arithmetic operations.
- **Reuse is Key**: To save data movement costs, hardware systems can leverage the reuse opportunities using dataflow mappings. Fortunately, DNN inference provided multiple areas for potential reuse.
- **Avoiding Computation Requires Orchestration**: Even seemingly obvious savings such as sparse inputs/weights require hardware design decisions that can introduce overhead and bottlenecks.

2.4.1 Pruning

The human brain synaptic connectivity follows a somewhat counter-intuitive trajectory. In the early embryonic stage, up to around two years of age, neuron connectivity explodes ten-fold to approximately 15,000 synapses per neuron. At around two, this growth stage ends, and the brain reverses course and begins synaptic pruning. The pruning away of synapses is so aggressive that by age 10, children have around 50% of the synapses they started with on their second birthday. Synaptic pruning continues, albeit somewhat slower, until early adulthood [Huttenlocher et al., 1979, Tau and Peterson, 2010].

This biological phenomenon is not yet fully understood, but the claimed parallels between computational neural networks and brain neurons have motivated researchers to explore the value of pruning neural networks as a form of learning and energy conservation. Seminal works explored this connection through the pruning of weights deemed unimportant [LeCun et al., 1990] and found performance can be maintained with a large number of weights removed. The motivation for doing so is predominantly in the avoidance of computation; a zero weight value need not be multiplied since the outcome is guaranteed to be zero.

Subsequent works have further developed the approach and found that pruning neural networks can be applied before [Lee et al., 2019, Lubana and Dick, 2020, Lee et al., 2019]

or during training [Lin et al., 2017], but is predominantly applied after training to convergence [Blalock et al., 2020, Han et al., 2015b].

We can describe the generalised pruning procedure as identifying a binary mask $M \in \{0,1\}^d$ where d is the number of parameters in the network. Let $\theta \in \mathbb{R}^d$ represent the parameter weights of the neural network. After identifying M, the new network is calculated taking the element-wise multiplication between parameter weights θ and M, $f(x|\theta \circ M)$.

How we identify these binary masks is of particular research interest, with the dominant approaches centring on pruning weights of low-magnitude [Han et al., 2015b] — a fast proxy for weight-importance — or, more explicitly, those that produce the smallest decrease in loss through examination of weight magnitudes scaled by the gradient [Molchanov et al., 2019b, 2016]. Others have looked at the disconnect between parameter reduction and energy reduction and opt to iteratively prune layer-wise but focus on layers which are the most energy draining first [Yang et al., 2017a].

A problem arises in determining how best to leverage the algorithmically induced sparsity. As we have been discussing, while the pursuit of fewer multiplications reduces computational resources, it would be far more beneficial if we could avoid these weights and their corresponding activations used in multiplication in the first place. Although libraries commonly used for DL optimisation CPUs and GPUs do not automatically identify and support sparsity there has been a move towards researching support [Gale et al., 2020]. The most accessible for the general practitioner has been using the PyTorch torch.sparse library, which leverages cuSPARSE [Naumov et al., 2010] and sparse storage formats such as compressed sparse column (CSC) and compressed sparse row (CSR), but this does require defining the architecture using such formats. Again, bespoke accelerator designs have been leading the way using compression formats such as RLE, CSR and CSC [Chen et al., 2017, Reagen et al., Dorrance et al., 2014, Han et al., 2016a].

An alternative option to consider in network pruning is to focus on structured components, such as entire filters, instead of individual weights. This way, there's no additional need for hardware optimisation, simplifying the process by just eliminating a channel in a layer. Several strategies for this type of structural pruning have been advanced, which leverage the Taylor series approximation to predict potential increases in loss when pruning occurs [Molchanov et al., 2016, 2019a]. Another effective strategy involves adjusting or 'regularising' the batch-normalisation scaling factor, and identifying filters with small norms for pruning [Liu et al., 2017].

An additional debate has surfaced into rethinking the value of pruning [Blalock et al., 2020], wherein the number of parameters left after pruning is not always competitive

with efficient architecture choices and benchmarking between pruning solutions not well-compared. However, efficient specialised hardware implementations are more able to make use of pruning induced sparsity [Reagen et al., Han et al., 2016a] as well as the activation sparsity observed due to ReLU non-linearities between layers [Sze et al., 2017] than the heterogeneous filter sizes used in architecture optimisations alone.

2.4.1.1 Lottery Ticket Hypothesis

The Lottery Ticket Hypothesis (LTH) [Frankle and Carbin, 2019] challenges the conventional belief that pruned networks can only achieve strong performance when coupled with optimised weights. Instead, the authors found that it's possible to identify a binary mask M, reinitialise the non-pruned weights to their original values, retrain the network while fixing M, and still maintain performance.

Let $\theta_0 \in \mathbb{R}^d$ denote the initial parameter values of a full network before any training or pruning. The LTH process can be described mathematically as follows:

1. Initial training: $\theta_0 \xrightarrow{\text{SGD}} \theta_t$, where θ_t represents the trained weights after t steps of stochastic gradient descent (SGD).
2. Pruning: Determine mask $M \in \{0,1\}^d$ based on θ_t, often by magnitude pruning where smaller weights are set to zero.
3. Apply mask: $\theta_p = \theta_t \circ M$, where $\circ$ denotes element-wise multiplication.
4. Reset to initial values: $\theta_r = \theta_0 \circ M$, resetting non-pruned weights to their initial values.
5. Retrain: $\theta_r \xrightarrow{\text{SGD}} \theta_f$, where θ_f represents the final weights after retraining.

Throughout steps 4 and 5, M remains fixed, ensuring that pruned weights (where $M_i = 0$) stay at zero. The LTH suggests that there exist sub-networks within the full network, defined by M, such that when the non-pruned weights are reset to their values in θ_0 and retrained with M fixed, the sub-network can achieve comparable performance to the original full network. Mathematically, this means: $f(x|\theta_f) \approx f(x|\theta_t)$, where $f(x|\theta)$ represents the network's output given input x and parameters θ.

With this seemingly simple setup, the authors of LTH achieved up to 95% pruning while maintaining the same performance as the pre-pruned network. The key aspect is that the non-pruned weights are always reset to their original initialisation values in θ_0 before retraining. This finding suggests that over-parametrisation of networks could be useful in allowing a larger space of options from which a sub-network can be

discovered during the training process. It also opens up potential research directions in identifying these pruned sub-networks early, which could significantly reduce the time and computational resources required for training.

The LTH has excited the research community, leading to numerous follow-up discoveries. These include the importance of using the largest magnitude or magnitude change coupled with zero masking [Zhou et al., 2019], the need for rewinding to a later iteration for more complex datasets [Frankle et al., 2019], the transferability of sub-networks to other classification problems [Morcos et al., 2019], and the applicability of lottery tickets beyond image classification problems [Yu et al., 2019, Girish et al., 2021, Chen et al., 2021].

The most pressing challenges include identifying the sub-networks without going through the arduous iterative train-prune-repeat cycle and understanding the implications of these sub-networks within the optimisation process. For our research, the pruning literature and LTH demonstrate the flexibility we have in inducing noise and constricting the optimisation process. The iterative nature of LTH directly influenced Weight Fixing Networks (Chapter 3), where we explore reforming the entire network to a pool of weights.

2.4.2 Quantisation

In the previous section, we looked at mapping the weights in DNN models to zero values to avoid memory reads and multiplications. While this approach can significantly reduce the computational burden, it is also essential to consider the efficiency of representing and processing the remaining non-zero weights. This is where *quantisation* comes into play.

Quantisation is a technique that aims to reduce the memory footprint and computational complexity of DNNs by representing weights with a reduced set of discrete values. By quantising the weights, we can store them using fewer bits per weight, leading to reduced memory usage and faster arithmetic operations. This is particularly important for deploying DNNs on resource-constrained devices, such as mobile phones or embedded systems, where memory and computational power are limited.

Moreover, quantisation can also lead to faster inference times, as the reduced precision of the weights allows for more efficient hardware implementations, such as using fixed-point arithmetic instead of floating-point operations. This can result in significant speedups, especially on specialised hardware.

Definition. Consider a network $\mathcal{N}$ parameterised by N weights $W = \{w_1, ..., w_N\}$. Quantising a network is the process of reformulating $\mathcal{N}' \leftarrow \mathcal{N}$ where the new network $\mathcal{N}'$ contains weights which all take values from a reduced pool of k cluster centres $C = \{c_1, ..., c_k\}$ where $k \ll N$. After quantisation, each of the connection weights in the original network is replaced by one of the cluster centres $w_i \leftarrow c_j$, $W' = \{w'_i | w'_i \in C, i = 1, \cdots, N\}$, $|W'| = k$, where W' is the set of weights of the new network $\mathcal{N}'$, which has the same topology as the original $\mathcal{N}$. Depending on the set-up, only a subset of the k cluster centres may be used within a particular layer or kernel channel. If performing layer-wise quantisation and the resulting k number of cluster centres within a particular layer is, say k=256, then we say this layer has been quantised to $\log_2 k = 8$-bits. Of course, this would only truly 8-bit if the values where symmetrical around a center point enabling a scale-shift mapping - we'll expand this discussion shortly, but first let's keep going with defining quantisation.

In the quantised network, each original weight w_i is replaced by a quantised value w'_i, where w'_i is chosen from the set of cluster centres C. The process of replacing the original weights with their quantised counterparts introduces a *quantisation error*, which measures the difference between the original and quantised weights. The quantisation error can be defined as: $Err(W, W') = \sum_{i=1}^{N} |w'_i - w_i| \cdot p(w_i, w'_i)$ where $p(w_i, w'_i)$ is the joint probability distribution of the original weight w_i and its quantised value w'_i.

This joint probability distribution captures the likelihood of a particular original weight being assigned to a specific cluster centre during the quantisation process. The quantisation error is the sum of the absolute differences between the original and quantised weights, weighted by their joint probabilities. Minimising this quantisation error is a key objective in designing effective quantisation schemes, as it helps to preserve the accuracy of the model while reducing its memory footprint and computational complexity.

In Figure 2.4, we show the distribution of a single pre-trained layer of a ResNet-18 model [He et al., 2016] trained on ImageNet. The simplest form of quantisation, shown on the top row, is linear quantisation. Here we take uniform points covering the entire range of weights and then assign these cluster values to the weights closest to each. Since the distribution of weights in a neural network tends to resemble a Gaussian distribution, using linear quantisation results in non-uniform cluster assignments and larger expected quantisation errors. An approach that offers reduced quantisation error is logarithmic quantisation, as shown in the bottom row of Figure 2.4. Here points are sampled to be powers-of-two, the assignment distribution is much closer to uniform over the relevant range, and the quantisation error is reduced. Using the L1-form for the quantisation error, we can see that the logarithmic quantisation gives us around a 20% error reduction following the weight distribution. Interestingly, this non-uniform

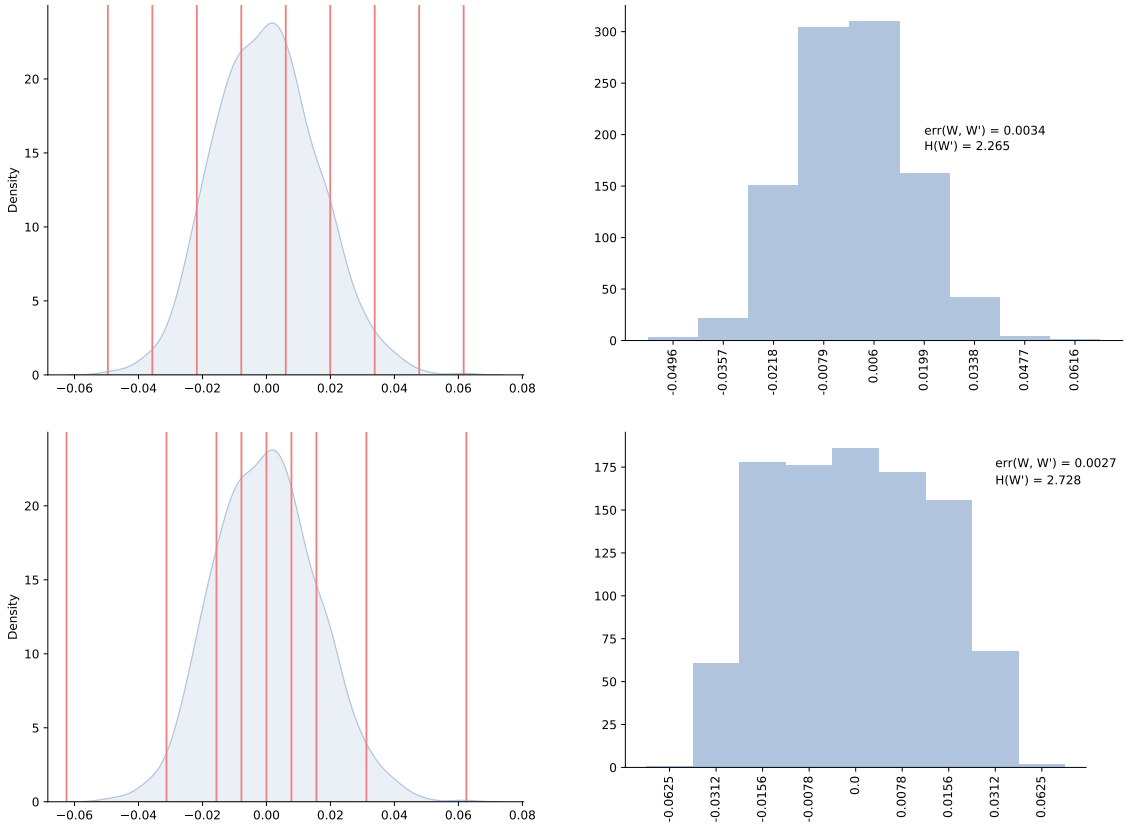


FIGURE 2.4: Comparing linear (top) and log (bottom) quantisation of the first layer of a pre-trained ResNet-18 model. In the left column we show the original layer weights with vertical lines indicating the cluster centres selected and on the right the distribution of weights post-quantisation.

sampling approach bears resemblance to the histogram equalisation observed in the fly's visual system, as described in Laughlin's work [Laughlin, 1981].

However, there is a small price for the lower error in the form of the weight-space entropy $H(W')$, which captures how much information is left on average in the network weights. This trade-off is an active consideration in all forms of network compression; the higher the fidelity of the compressed version to its original, the less *compressed* the network is. Fortunately, the notion of fidelity has a strict meaning in the quantisation error but does not automatically translate into the fidelity of task performance. In other words, we can alter a DNN such that the transformed weights do not closely resemble the original but still maintain task performance, meaning there is a many-to-one mapping of weight sets to tasks. Quantisation could be thought of as translating one weight-set with high task performance but high weight-space entropy to another with the same or similar performance but with much lower weight-space entropy.

One common form of quantisation, referred to as *clip-and-scale*, maps the weight w_i to a cluster $c_k = quant(w_i; s, z, b)$ as shown in Equation 2.4, where s, z, and b represent the scale factor, zero-point, and desired bit-width, respectively - shown in Figure 2.5.

FIGURE 2.5: Clip-and-scale quantisation for a single cycle.

$$clamp(w_i; a, c) = \begin{cases} a, & \text{if } w_i < a \\ w_i, & \text{if } a \leqslant w_i \leqslant c \\ c, & \text{if } w_i > c \end{cases} \tag{2.3}$$

$$quant(w_i; s, z, b) = s[clamp(\lfloor \frac{w_i}{s} \rceil + z, 0, 2^b - 1) - z] \tag{2.4}$$

The main benefit of clip-and-scale quantisation is that it allows for the use of fixed-point arithmetic instead of floating-point operations. Fixed-point arithmetic is more energy-efficient and faster than floating-point arithmetic, as it requires fewer bits to represent numbers and can be implemented using simpler hardware. This is particularly important for edge devices and specialised hardware accelerators, where energy efficiency and computational speed are critical.

The scaling factor s can be learnt channel-wise [Jacob et al., 2018, Zhang et al., 2018] or layer-wise in separate formulations, resulting in different channels or layers having a diverse pool of codebooks for network weights, activations, and gradients. This allows for a more fine-grained quantisation approach, potentially leading to better preservation of the network's accuracy. While the bit-width b is traditionally fixed for all layers and channels, some works have explored varying b between layers [Das et al., 2018, Wang et al., 2019, Huang et al., 2021], and others have demonstrated that you can train a model once and select the quantisation precision based on the downstream device requirements [Jin et al., 2020]. This flexibility is beneficial for deploying models

on a variety of hardware platforms with different computational capabilities and energy constraints.

However, clip-and-scale quantisation methods also have some weaknesses. The clipping operation can lead to information loss, as values outside the clipping range are truncated. This can result in a degradation of the network's accuracy, especially if the clipping range is not carefully chosen. Furthermore, the scaling factor and zero-point need to be stored alongside the quantised weights, adding some overhead to the model's memory footprint. This is particularly true the more fine-grained you quantised with more scaling factors for different groups of weights to attempt to mitigate any accuracy drops.

This is where techniques like clustering-based quantisation [Stock et al., 2020, Tartaglione et al., 2021, Wu et al., 2018]. attempt to do better, using shared codebooks across the entire network. These approaches aim to reduce the overall memory footprint and minimise the data reads required. These methods use clustering techniques to cluster the weights and fix the weight values to their assigned group cluster centroid. These weights are stored as codebook indices, allowing for compressed representation methods such as Huffman encoding to further compress the network.

The work by [Wu et al., 2018] is of particular interest since both the motivation and approach are related to our Weight Fixing Network method (Chapter 3). The authors use a *spectrally relaxed* k-means regularisation term to encourage the network weights to be more amenable to clustering. They focus on a filter-row codebook inspired by the row-stationary dataflow used in some accelerator designs [Chen et al., 2017]. However, their formulation is explored only for convolution, and they restrict clustering to groups of weights (filter rows) rather than individual weights due to computational limitations, as recalibrating the k-means regularisation term is expensive during training.

Taking quantisation to the extreme, binary neural networks [Courbariaux et al., 2015] reduce the weight-value precision to a single mantissa, +1 or -1. This allows multiplication in the MAC operations to be reduced to addition and subtraction. Subsequent work [Hubara et al., 2016] quantises the activation output to binary values, which turns the MAC operations into an XNOR operation. This reduction comes at the cost of accuracy for all, but the simplest of problems and tweaks are required to make these networks more viable — such as allowing full precision in the last layers [Hubara et al., 2016].
An interesting observation is that the precedent set by binary neural networks of allowing the first and last layers to have full precision is now a tweak prominent in almost all general quantisation works reviewed. This practice seems to be an ad-hoc way to mitigate the accuracy loss caused by the limited capacity of heavily quantised models to memorise training examples. The long tail theory proposed by [Feldman and

Zhang, 2020] provides a compelling explanation for the necessity of memorisation in achieving high accuracy on long-tailed data distributions. The theory suggests that memorisation is crucial for learning from rare and atypical examples in the ”long tail” of the data distribution, which can be statistically indistinguishable from useless examples such as outliers and mislabelled data points. Empirical results on several standard benchmarks provide quantitative and visually compelling evidence for the long tail theory. The implications of this theory are significant, as techniques that limit memorisation, such as quantisation, may have a disproportionate effect on under-represented sub-populations. We will later quantify the impact of this compression relaxation by comparing the number of unique parameters to the range of values they can take. This will show that increasing the precision of the floating-point operations can compensate for the reduction in unique parameters, thereby optimising data-movement costs. The idea is that you can reduce the data-movement costs at the expense of increasing the precision at which you need to apply floating-point operations. Relaxations may be needed, but there are choices of where to make them that favour downstream hardware fundamentals.

2.4.3 Architecture Improvements

All the algorithmic methods discussed so far assume the architecture itself is fixed and energy-savings are made by altering the weight values themselves, another branch of energy-saving approaches instead looks to make saving through adapting the architecture itself.

Although the number of parameters required are usually fewer in CNN layers than that of fully-connected layers, the convolution operator dominates the MAC operations. These computational costs grow with the product of the number of channels and size of filters. Decomposition of filters is therefore one route to a reduction, an NxN convolution can be decomposed into two 1-D convolutions of size 1xN and Nx1 for example as long as the 2-D filter is separable. 1x1 convolutional layers can also be used to reduce the number of channels before the larger filters are applied — known as a bottleneck.
Motivated by this, the separable convolution [Chollet, 2017a] decomposes the convolution operation across a large filter-size and number of channels into a 1x1 pointwise convolution followed by the larger spatial convolution. The benefit of doing so is that the pointwise convolution reduces the channels such that the spatial filter can be applied with a smaller computational cost. GoogLeNet [Szegedy et al., 2014] uses these layers in achieving what was state-of-the-art performance at the time, whilst reducing the number of MAC operations from 854 million to 358 million.

These separable convolutions have since been further utilised in SqueezeNet [Iandola et al., 2016] — which reduced pre-trained AlexNet [Krizhevsky et al., 2012] by 50x to a 0.5MB model using multiple 1x1 convolution followed by spatial filter *fire* modules. MobileNet [Howard et al., 2017], which adds, additionally, a latency regulariser, demonstrates good performance using networks small enough to run on a mobile device. It should be noted that simply having fewer weights is not a proxy for resource-constrained applications, SqueezeNet, for example, consumes more energy than AlexNet despite having far fewer weights [Sze et al., 2017]. As we discussed previously, it is DRAM reads which dominate the energy-costs and so heterogeneous filter sizes and deeper networks (prompting lots of weight value fetching) are not easily optimised for reuse avoiding re-reads from off-chip memory.

2.4.4 Architecture Search

The current workflow for DNN architecture design primarily requires heuristically driven trial-and-error layer and loss combinations to be hand-coded and experimentally driven decisions on *good* architectures. Neural architecture search (NAS) is a sub-field of research concerned with identifying, in an automated way, optimal architectures [Zoph and Le, 2016, Pham et al., 2018, Elsken et al., 2019, Tan and Le, 2019]. Successful approaches have used these techniques with additional search constraints [Tan et al., 2019, Lin et al., 2020a, Banbury et al., 2021], favouring resource-efficient solutions.

2.4.5 Student-teacher

Rather than alter weights to come from a pool of values or find resource-lite architectures, student-teacher methods (also known in the literature as knowledge-distillation) propose ways where a *teacher* neural network can help teach a *student* neural network and, in doing so, can condense the knowledge required to complete the task being learned such that the student network can be much smaller than the teacher network [Gou et al., 2021].

This idea, pioneered by the work of [Hinton et al., 2015], carries out the transfer of knowledge from a fully trained teacher network to the student network. These methods typically do so by adjusting the loss function such that the student is trained to predict the target and match its logits with the teacher, given the same datapoint, thereby encouraging the student network towards valuable representations of the input data but using a smaller model.

2.5 Algorithm:Hardware co-design

The discussed choice of techniques used to reduce deep learning models available to practitioners is not an either-or decision. Deep compression [Mao and Dally, 2016b] demonstrates the compounding nature of combining resource reductionist approaches. The authors apply iterative pruning to a network during training, followed by layer-wise quantisation. Next, they force further weight-sharing, iteratively applying clustering techniques to reduce the number of distinct weights in the network allowing for a code-book look-up of a smaller set of weights which can then be further compressed using huffman encoding. They demonstrate the compounding nature of these efficiency gains reducing the storage cost of VGG-16 by 49X.

This approach was later further optimised in custom hardware [Han et al., 2016a] which utilise a row variant of compressed sparse column format [Yoshizawa and Takahashi, 2012] with an array of processing elements fitted with SRAM to utilise both the weight-matrix and activation sparsity achieved in compression. This combination of compression at the architecture level coupled with custom hardware resulted in a 2,700x energy saving compared with a mobile GPU running a compressed AlexNet [Krizhevsky et al., 2012] model.

This is one such example of the promise of the field. Whilst this thesis is predominantly focussed on algorithmic approaches to energy saving, we hope to make these better informed by starting with the hardware fundamentals and working our way up to novel algorithmic approaches, co-designed with hardware.

Chapter 3

Weight Fixing Networks

Having identified the challenges faced by accelerator designs in arranging components and producing efficient data mapping schemes to maximise parameter re-use, in this chapter we look to develop an algorithmic procedure to enable more efficient downstream hardware design options.

As we explored in the background section, the dominant energy cost in deep neural network (DNN) inference comes from data movement, particularly from off-chip DRAM to on-chip memory. We saw that arithmetic operations are orders of magnitude less costly than these memory accesses. This fundamental insight motivates our approach in this chapter.

We take the view that reducing the number of unique weights coupled with reducing overall network weight-space entropy gives a measurable goal from an algorithmic perspective that allows for downstream hardware flexibility in design for re-use opportunities. In the best-case scenario, each unique weight would be read once and be used for all the computations it is involved in. This is, of course, highly unlikely, since it would require the storage of the entire network (all of the unique weights) close to computation. However, as the number of unique weights is reduced, so is the cost to store the network, paying tribute to its utility as an algorithmic objective.

The weight-space entropy,

$$H(W) = -\sum_{w_i \in W} p(w_i) \ln(p(w_i)), \tag{3.1}$$

refers to the average number of bits of information within each of the parameters in a network and represents the theoretically fewest number of bits of information to

losslessly compress the weight-space Shannon [1948a]. Referring back to Figure 2.4, $p(w_i)$ is the probability that weight value w_i was picked at random from all unique weights in the network.

Minimising this value has two benefits from a practical standpoint, and algorithmic compression schemes have been focused on getting as close to this limit as possible. A network with entropy $H(W)$ can be stored using $NH(W)$ bits (where N is the number of weights in the network), and so minimising the weight-space entropy is a close proxy for minimising the storage size requirements for the network. Secondly, in minimising the entropy of the weight-space, we are focusing on just a few weights around high probability regions with a large number of seldom seen outlier weights. The high probability weights will be re-used often in a forward pass and are therefore prime candidates for keeping close to computation to reduce data movement costs as previously discussed.

Recall from our discussion on dataflow mappings that weight-stationary designs, such as those used in Tensor Processing Units (TPUs), aim to keep weights in local memory and stream in activations. By reducing the number of unique weights, we can potentially enhance the effectiveness of such designs, allowing for more weights to be stored locally and reducing the need for repeated DRAM accesses.

Furthermore, our exploration of quantisation techniques demonstrated the potential for reducing the precision of weights without significant loss in accuracy. The Weight Fixing Networks approach we'll introduce in this chapter can be seen as an extreme form of quantisation, where we aim to drastically reduce the number of unique weight values across the entire network.

In this chapter, we take on the challenge of compressing the entire network using a single codebook and demonstrate superior lossless compression using a few novel (and some well-trodden) tricks: a view of compression as *relative* distance minimisation, a novel regularisation term to encourage low-entropy network encodings, and a focus on optimising a whole-network quantisation codebook, rather than per-layer. The pipeline, which we call Weight Fixing Networks (WFN), achieves lossless compression using 50x fewer unique weights, half the weight-space entropy, and at least a 14% improvement in storage compressibility when compared with state-of-the-art (SOTA) quantisation and weight clustering approaches with no reduction in classification accuracy.

The idea is that with a single codebook defining the entire network, we will maximise the amount of re-use of the parameters and therefore enable hardware designs to focus resources on storing the re-used parameters close to computation - reducing off-chip memory re-use/cache misses.

This approach builds on insights from hardware accelerator design, quantisation, pruning, and information theory. Our goal is to develop an algorithmic method that can produce networks with drastically reduced numbers of unique weights, lower weight-space entropy, and maintained task performance – all while keeping in mind the potential for efficient hardware implementation.

3.1 Single Codebook

Seminal works quantising modern iterations of neural networks proposed that some layers should be left at full precision whilst the rest of the network was quantised [Hubara et al., 2016]. As we have previously discussed the original formulation was mooted originally for binary networks (where the weights take on only one of two values) and guided by necessity. Since binarised networks are restricted to only the values: $\{-1, 1\}$ task performance suffers an intolerable amount if one or more layers are not left at full precision. The first layer tends to have a low parameter count (due to a small channel input) and so were the first to be relaxed to full precision [Hubara et al., 2016]. Shortly after, XNOR-Net [Rastegari et al., 2016] extended the non-quantised layers to also include the final layer. Further developments in neural network design have seen the batch-normalisation layer included in most SOTA convolution architectures but the parameters involved in these calculations are not considered for quantisation. These relaxations have become a universal trend not just for binarisation approaches but for higher-bit-width quantisation in general [Yuhang Li, Xin Dong, 2020, Jung et al., 2019a, Zhang et al., 2018, Zhou et al., 2016a, Yamamoto, 2021]. Our main argument is that this relaxation contributes significant computational overhead and is unnecessary outside of the highly restrictive binary neural network case.

3.1.1 Unquantised Elements

In Figure 3.1, we take a look at how much of the network remains unquantised with this set-up for popular models. Taking ResNet-50 as an example, over 2 million weights are left unquantised, accounting for 8% of all parameters in the network. This uncompressed network component translates into a computational overhead in hardware, increased floating-point operations (FLOPs), and, crucially, higher data movement costs due to memory reads.

Another prevalent trend found in quantisation methods is to use different codebooks for each layer. This is justified by the *you-only-pay-once* argument, where, since each layer is only read once per inference calculation, the cost of reading the codebook for

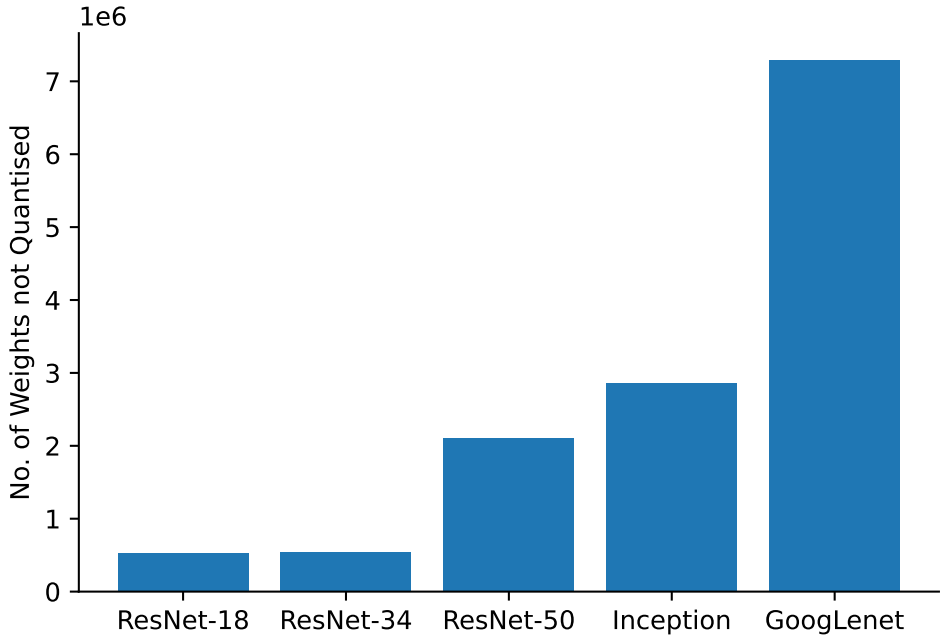


FIGURE 3.1: We count the number of parameters that are left unquantised by the "status-quo" approaches that leave the first, last, linear and batch-norm layers to be 8-bit or full precision. Each of these parameters incurs energy-hungry data movement costs for accelerators.

each layer is small. This, however, increases the overall weight-space entropy, which in turn results in sub-optimal compression and less parameter re-use.

Using multiple codebooks and allowing parts of the network to remain unquantised has real implications for deep learning accelerator designs. Recalling our previous discussion, data movement costs are a function of parameter re-use, the delay in re-use and filter shape effects [Sze et al., 2017, 2020]. These costs are difficult to reduce in von Neumann architectures [Li et al., 2015, Sebastian et al., 2020] and are a core consideration of accelerator designs. Specialised hardware looks to distributing computation across processing elements (PE's) which typically contain both a computational and a memory unit. The reading of data from the memory unit is small [Chen et al., 2020, Sze et al., 2020] but the number of PE's is limited ($\ll 1000$ is typical). Here, we see why not having a whole network codebook and/or not quantising every layer is costly; less re-use leads to increased data movement costs since each unique value needs reading at least once. This single read per weight is still only the best-case scenario since if the network has a larger number of unique parameters than PE's, then parameters that require re-use may need to be read and then re-read multiple times.

3.1.2 WFN Objectives

Given these observations, we take the view that fewer *unique* weights across the entire network to be a worthy objective. Fewer weights – whilst fixing the network topology and the total number of parameters – means that more weights are re-used more often. This additional re-use gives more opportunity to dataflows to maintain often-used weights in PE's, saving data movement costs. To save data movement costs further, all network weights could be quantised into a shared codebook which would save between-layer re-reading. To further the compression capability, we note that it would be desirable if the distribution of the unique weights were concentrated around a handful of values. The high probability density weights would then be used more often and reliably stored inside PE's, saving both the cost of overwriting these weight values and re-fetching them when needed later. Finally, from a computational perspective, not all multiplications are created equal. Integer powers-of-two, for example, can be implemented as simple bit-shifts. Focussing the weights used most to these values offers potential further energy reductions. Putting these three requirements together: few unique weights; a low-entropy encoding with a distribution of weights highly concentrated around a tiny subset of values; and a focus on powers-of-two values for weights — all motivated to reduce the data movement costs — we present the outline of WFN.

3.1.3 Overview

Our work's overarching objective is to transform a network comprising many weights of any value (limited only by value precision) to one with the same number of weights but just a few unique values and concentrate the weights around an even smaller subset of weights. Rather than selecting the unique weights a priori, we let the optimisation guide the process in an iterative *cluster-then-train* approach. We cluster an ever-increasing subset of weights to one of a few cluster centroids in each iteration. We map the pre-trained network weights to these cluster centroids, which constitute a pool of unique weights. The training stage follows standard gradient descent optimisation to minimise performance loss with two key additions. Firstly, only an ever decreasing subset of the weights are *free* to be updated. We also use a new regularisation term to penalise weights with large relative distances to their nearest clusters. We iteratively cluster subsets of weights to their nearest cluster centre, with the way we determine which subset to move a core component of our contribution.

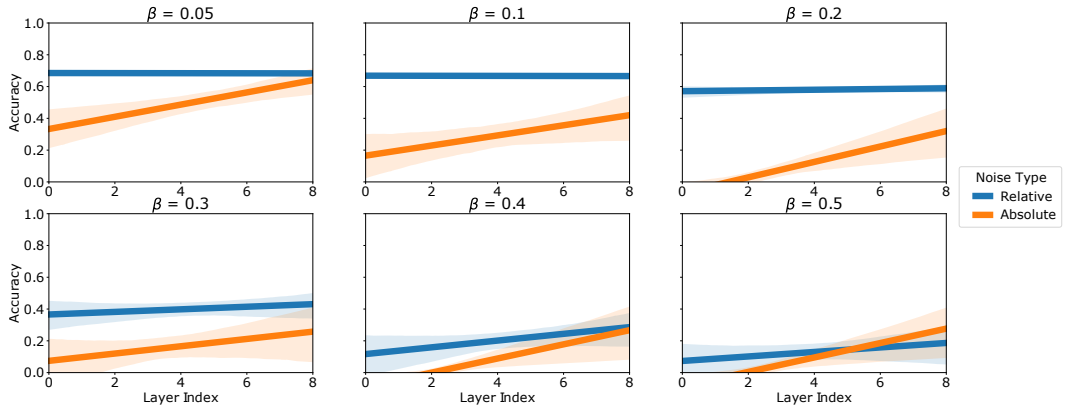


FIGURE 3.2: We explore adding relative vs absolute noise to each of the layers (x-axis). The layer index indicates which layer was selected to have noise added. Each layer index is a separate experiment with the 95% confidence intervals shaded.

3.1.4 Small Relative Distance Change.

Rather than selecting subsets with small Euclidean distances to cluster centres, or those that have small magnitude [Zhou et al., 2017], we make the simple observation that the *relative* – as opposed to absolute – weight change matters. We find that the tolerated distance δw_i we can move a weight w_i when quantised depends on the relative distance $|(\delta w_i / w_i)|$. When the new value $w_i + \delta w_i = 0$ — as is the case for pruning methods — then the magnitude of the weight *is* the distance. However, this is not the case more generally. We demonstrate the importance of quantising with small relative changes using simple empirical observations. Using a pre-trained ResNet-18 model, we measure changes to network accuracy when adding relative vs absolute noise to the layers' weights and measure the accuracy change. For relative (multiplicative) noise we choose a scale parameter $\beta|w_i^l|$ for each layer-l weight w_i^l, and set $w_i^l \leftarrow w_i^l + \beta|w_i^l|\varepsilon$, $\varepsilon \sim \mathcal{N}(0,1)$. In contrast, additive noise perturbations, all weights w_i^l are perturbed by the mean absolute value of weights $\overline{|w^l|}$ in layer l scaled by β: $w_i^l \leftarrow w_i^l + \beta\overline{|w^l|}\varepsilon$.

We run each layer-β combination experiment multiple times – to account for fluctuation in the injected noise – and present the results in Figure 3.2. Even though the mean magnitude variation of noise added is the same, noise relative to the original weight value (multiplicative noise) is much better tolerated than absolute (additive noise). Since moving weights to quantisation centres is analogous to adding noise, we translate these results into our approach and prioritise clustering weights with small relative distances first. We find that avoiding significant quantisation errors requires ensuring that $\frac{|\delta w_i|}{|w_i|}$ is small for all weights. If this is not possible, then performance could suffer. In this case, we create an additional cluster centroid in the vicinity of an existing cluster to reduce this relative distance. Our work also challenges the almost universal trend in the literature [Yuhang Li, Xin Dong, 2020, Jung et al., 2019a, Zhang et al., 2018, Zhou et al.,

2016a, Yamamoto, 2021, Oh et al., 2021a] of leaving the first and last layers either at full precision or 8-bit. Instead, we attempt a full network quantisation. The cost of not quantising the first layer – which typically requires the most re-use of weights due to the larger resolution of input maps – and the final linear layer – which often contains the largest number of unique weight values – is too significant to ignore.
With multiple stages of training and clustering, we finish with an appreciably reduced set of unique weights. We introduce a regularisation term that encourages non-uniform, high probability regions in the weight distribution to induce a lower-entropy weight-space. The initial choice of cluster centroids as powers-of-two helps us meet our third objective – energy-saving multiplication. Overall we find four distinct advantages over the works reviewed:

- We assign a cluster value to *all weights* — including the first and last layers.
- We emphasise a low entropy encoding with a regularisation term, achieving entropies smaller than those seen using 3-bit quantisation approaches – over which we report superior performance.
- We require no additional layer-wise scaling, sharing the unique weights across all layers.
- WFN substantially reduces the number of unique parameters in a network when compared to existing SOTA quantisation approaches.

3.2 Method

3.2.1 Method Outline.

WFN is comprised of T *fixing iterations* where each iteration $t \in T$ has a training and a clustering stage. The clustering stage is tasked with partitioning the weights into two subsets $W = W^t_{\text{fixed}} \cup W^t_{\text{free}}$. W^t_{fixed} is the set of weights set equal to one of the cluster centre values $c_k \in C$. These *fixed* weights $w_i \in W^t_{\text{fixed}}$ are not updated by gradient decent in this, nor any subsequent training stages. In contrast, the *free-weights* denoted by W^t_{free} remain trainable during the next training stage. With each subsequent iteration t we increase the proportion $p_t = \frac{|W^t_{\text{fixed}}|}{|W|}$ of weights that take on fixed cluster centre values, with $p_0 < p_1 \ldots < p_T = 1$. By iteration T, all weights will be fixed to one of the cluster centres. The training stage combines gradient descent on a cross-entropy classification loss, along with a regularisation term that encourages tight-clusters, in order to maintain lossless performance (i.e no drop in performance compared to the baseline trained network) as we fix more of the weights to cluster centres.

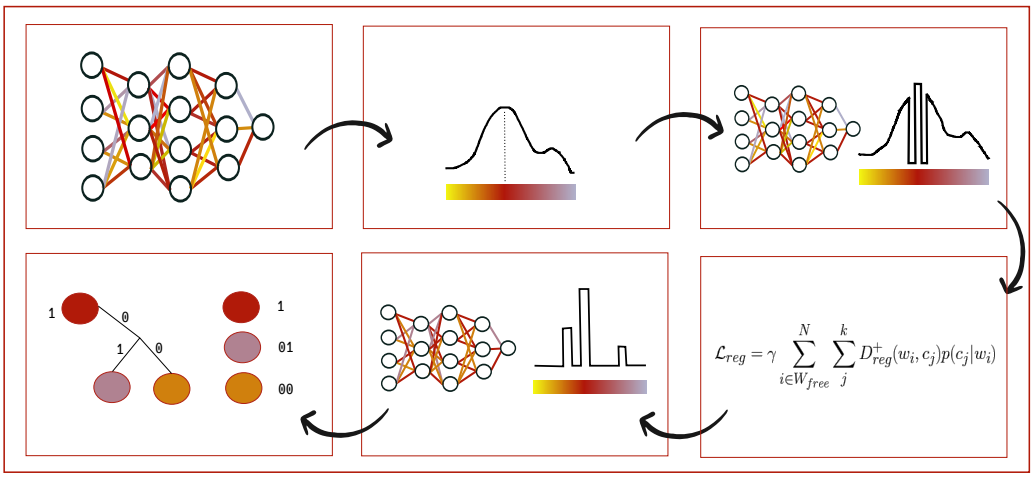


FIGURE 3.3: **The WFN pipeline** We start with a fully-trained converged model and assign k initial cluster centroids based on relative distance changes between the original weights and their closest cluster centres. We then retrain the network for a few epochs with an additional regularisation term that encourages the weights towards their closest cluster centroids. Weights that are subsequently close enough to their closest cluster, without a distance threshold breach, can then be fixed to their closest cluster value. If not enough weights can be fixed without the threshold being breached, k can increase, adding a new cluster centre. The process is repeated until the network is reformulated and uses just a few unique weights. By construction, the weight distribution will be skewed for efficient low-entropy Huffman encoding.

Algorithm 1: Clustering Np_t weights at the t^{th} iteration.

while $|W^{t+1}_{\text{fixed}}| \leq Np_t$ **do**
  $\omega \leftarrow 0$
  $\text{fixed}_{\text{new}} \leftarrow [\,]$
  while $\text{fixed}_{\text{new}}$ *is empty* **do**
    Increase the order $\omega \leftarrow \omega + 1$
    for each $i = 1 \ldots, |W^{t+1}_{\text{free}}|$
      $c^{\omega}_{*}(i) \leftarrow \min_{c \in \widetilde{C}^{\omega}} D^{+}_{\text{rel}}(w_i, c)$
    for each cluster centre $c^{\omega}_{k} \in \widetilde{C}^{\omega}$
      $n^{\omega}_{k} \leftarrow \sum_i \mathbb{I}[c^{\omega}_{k} = c^{\omega}_{*}(i)]$
    $k^{*} \leftarrow \arg\max_k n^{\omega}_{k}$
    Sort: $[w'_1, \ldots, w'_N] \leftarrow [w_1, \ldots, w_N]$, $w'_i = w_{\pi(i)}$, π permutation
      where $D^{+}_{\text{rel}}(w'_i, c^{\omega}_{k*}) < D^{+}_{\text{rel}}(w'_{i+1}, c^{\omega}_{k*})$
    $i \leftarrow 1$, mean $\leftarrow D^{+}_{\text{rel}}(w'_1, c^{\omega}_{k*})$
    while mean $\leq \delta^t$ **do**
      $\text{fixed}_{\text{new}} \leftarrow w'_i$
      mean $\leftarrow \frac{i}{i+1} *$ mean $+ \frac{1}{i+1} D^{+}_{\text{rel}}(w'_{i+1}, c^{\omega}_{k*})$
      $i \leftarrow i + 1$
  Assign all the weights in $\text{fixed}_{\text{new}}$ to cluster centre $c^{\omega}_{*}(i)$, moving them from W^{t+1}_{free} to W^{t+1}_{fixed}

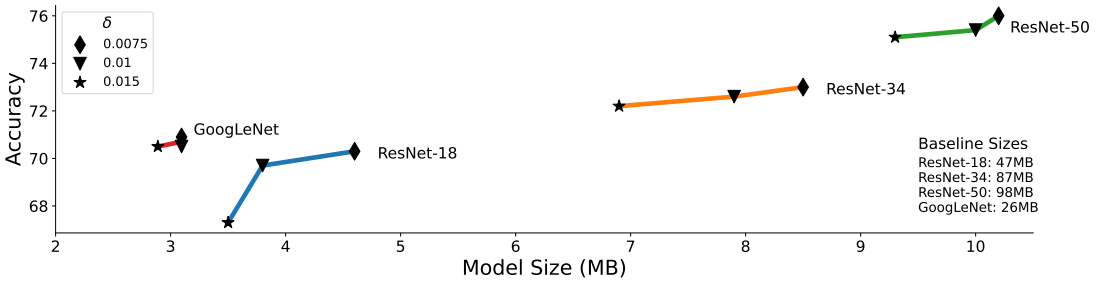


FIGURE 3.4: The accuracy vs model size trade-off can be controlled by the δ parameter. All experiments shown are using the ImageNet dataset, accuracy refers to top-1.

3.2.2 Clustering Stage.

In the clustering stage, we work backwards from our goal of minimising the relative distance travelled for each of the weights to determine which values cluster centres $c_i \in C$ should take. For a weight $w_i \in W$ and cluster centre $c_j \in C$ we define a relative distance measure $D_{\text{rel}}(w_i, c_j) = \frac{|w_i - c_j|}{|w_i|}$. To use this in determining the cluster centres, we enforce a threshold δ on this relative distance, $D_{\text{rel}}(w_i, c_j) \leq \delta$ for small δ. We can then define the cluster centres $c_j \in C$ which make this possible using a simple recurrence relation. Assume we have a starting cluster centre value c_j, we want the neighbouring cluster value c_{j+1} to be such that if a network weight w_i is between these clusters $w_i \in [c_j, \frac{c_{j+1}+c_j}{2}]$ then $D_{\text{rel}}(w_i, c_j) \leq \delta$. Plugging in $\frac{c_{j+1}+c_j}{2}$ and c_j into D_{rel} and setting it equal to δ we have:

$$\frac{|\frac{c_{j+1}+c_j}{2} - c_j|}{\frac{c_{j+1}+c_j}{2}} = \delta, \text{ leading to } c_{j+1} = c_j(\frac{1+\delta}{1-\delta}),\ 0 < \delta < 1, \tag{3.2}$$

a recurrence relation that provides the next cluster centre value given the previous one. With this, we can generate all the cluster centres given some boundary condition $c_0 = \delta_0$. δ_0 is the lower-bound cluster threshold, and all weights w_i for $|w_i| < \delta_0$ are set to 0 (pruned). This lower bound serves two purposes: firstly, it reduces the number of proposal cluster centres which would otherwise increase in density around zero with a reciprocal spacing, and additionally, the zero-valued weights make the network more sparse. This will allow sparsity-leveraging hardware to avoid operations that use these weights, reducing the computational overhead. As an upper-bound, we stop the recurrence once a cluster centre is larger than the maximum weight in the network, $\max_j |c_j| \leq \max_i |w_i|,\ w_i \in W, c_j \in C$.

3.2.3 Generating the Proposed Cluster Centres.

Putting this together, we have i) a starting point $c_0 = \delta_0$, ii) a recurrence relation to produce cluster centres given c_0 that maintains a relative distance change when weights

are moved to their nearest cluster centre, and iii) a centre generation stopping condition $c_j \leq \max_{i \in W} |w_i|$, $c_j \in C$. We use the δ_0 value as our first proposed cluster centre c_0 with the recurrence relation generating a proposed cluster set of size s. Since all these values will contain only positive values, we join this set with its negated version along with a zero value to create a proposal cluster set $C^S = \{a(\frac{1+\delta}{1-\delta})^j \delta_0 \mid j = 0, 1 \cdots s;\ a = +1, 0, -1\}$ of size $2s + 1$.

To account for the zero threshold δ_0 and for ease of notation as we advance, we make a slight amendment to the definition of the relative distance function $D_{\text{rel}}(w_i, c_j)$:

$$D^+_{\text{rel}}(w_i, c_j) = \begin{cases} \frac{|w_i - c_j|}{|w_i|}, & \text{if } |w_i| \geq \delta_0 \\ 0 & \text{otherwise.} \end{cases} \tag{3.3}$$

3.2.4 Reducing k with Additive Powers-of-two Approximations.

Although using all of the values in C^S as centres to cluster the network weights would meet the requirement for the relative movement of weights to their closest cluster to be less than δ, it would also require a large number of $k = |C^S|$ clusters. In addition, the values in C^S are also of full 16-bit precision, and we would prefer many of the weights to be powers-of-two for ease of multiplication in hardware. With the motivation of reducing k and encouraging powers-of-two clusters whilst maintaining the relative distance movement where possible, we look to a many-to-one mapping of the values of C^S to further cluster the cluster centres. Building on the work of others Zhou et al. [2017], Yuhang Li, Xin Dong [2020], we map each of the values $c_i \in C^S$ to their nearest power-of-two, $\text{round}(c_i) = \text{sgn}(c_i) 2^{\lfloor \log_2(|c_i|) \rceil}$ where $\lfloor \cdot \rceil$ represents the rounding operation (rounding up if the fractional part is greater than or equal to 0.5, rounding down otherwise), and, for flexibility, we further allow for *additive* powers-of-two rounding. With additive powers-of-two rounding, each cluster value can also come from the sum of powers-of-two values (b-bit) up to order ω where the order represents the number of powers-of-two that can contribute to the approximation. We map our proposal set C^S to a ω-order approximation where each of the clusters $c_k \in C^S$ are written as $c_k = \sum_{j=1}^{\omega} r_j$, $r_j \in \{-\frac{1}{2^b}, \ldots, -\frac{1}{2^{j+1}}, -\frac{1}{2^j}, 0, \frac{1}{2^j}, \frac{1}{2^{j+1}}, \ldots \frac{1}{2^b}\}$. The $\lfloor \cdot \rceil$ notation represents rounding to the nearest integer value. We do so using Algorithm 2. Figure 3.5 demonstrates how the values of C^S are rounded given different orders.

3.2.5 Minimalist Clustering.

We are now ready to present the clustering procedure for a particular iteration t, which we give the pseudo-code for in Algorithm 1. We start the iteration with $\omega = 1$ and a set

Algorithm 2: Determining possible clusters

Input: The full precision proposal set: C^S, allowable relative distance: δ, pow2 rounding function: $round(x) = sgn(x)2^{\lfloor \log_2(x) \rceil}$
Output: An order ω precision cluster set: $\widetilde{C}^\omega$
$\widetilde{C}^\omega \leftarrow [\,]$
for $c_k \in C^S$ **do**
  $c'_k = round(c_k)$
  for $i = 0 \rightarrow \omega$ **do**
    $\delta_{c_k} \leftarrow c_k - c'_k$
    if $|\delta_{c_k}| \geq \delta c_k$ **then**
      $c'_k \leftarrow c'_k + round(\delta_{c_k})$
    end
  end
  $\widetilde{C}^\omega \leftarrow \widetilde{C}^\omega \cup \{c'_k\}$
end

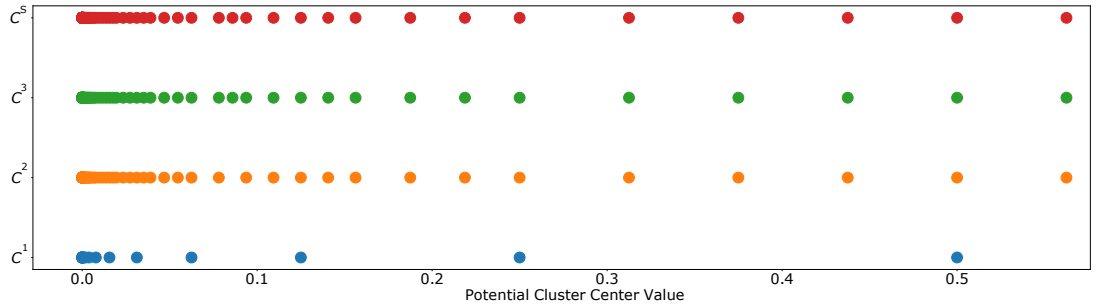


FIGURE 3.5: Approximating clusters $c_k \in C^S$ with different orders for $b = 7$

of weights not yet fixed W^t_{free}. For the set of cluster centres $\widetilde{C}^\omega$ of order ω, let $c^\omega_*(i) = \min_{c \in \widetilde{C}^\omega} D^+_{\text{rel}}(w_i, c)$ be the one closest to weight w_i. $n^\omega_k = \sum_i \mathbb{I}[c^\omega_k = c^\omega_*(i)]$ counts the number of weights assigned to cluster centre $c^\omega_k \in \widetilde{C}^\omega$, where the indicator function $\mathbb{I}[x]$ is 1 if x is true and 0 otherwise. Let $k^* = \arg\max_k n^\omega_k$ so that $c^\omega_{k^*}$ is the modal cluster. For the cluster k^* let permutation π of $\{1, \ldots, N\}$ that maps $w_i \mapsto w'_{\pi(i)}$, be such that the sequence $(w'_1(k^*), w'_2(k^*), \ldots, w'_N(k^*))$ is arranged in ascending order of relative distance from the cluster $c^\omega_{k^*}$. In other words, $D^+_{\text{rel}}(w'_i(k^*), c^\omega_{k^*}) \leq D^+_{\text{rel}}(w'_{i+1}(k^*), c^\omega_{k^*})$, for $i = 1, \ldots, (N-1)$. We choose n to be the largest integer such that:

$$\sum_{i=1}^{n} D^+_{\text{rel}}(w'_i(k^*), c^\omega_{k^*}) \leq n\delta, \text{ and } \sum_{i=1}^{n+1} D^+_{\text{rel}}(w'_i(k^*), c^\omega_{k^*}) > (n+1)\delta, \tag{3.4}$$

and define $\{w'_1, w'_2, \ldots, w'_n\}$ to be the set of weights to be fixed at this stage of the iteration. These are the weights that can be moved to the cluster centre $c^\omega_{k^*}$ without exceeding the average relative distance δ of the weights from the centre. The

corresponding weight indices from the original network $\mathcal{N}$ are in $\{\pi^{-1}(1), \ldots, \pi^{-1}(n)\}$, and called $\text{fixed}_{\text{new}}$in the algorithm. If there are no such weights that can be found, *i.e.*, for some cluster centre l^*, the minimum relative distance $D^+_{\text{rel}}(w'_1(l^*), c_{l^*}) > \delta$, the corresponding set $\text{fixed}_{\text{new}}$ is empty. In this case, there are no weights that can move to this cluster centre without breaking the δ constraint and we increase order $\omega \leftarrow \omega + 1$ to compute a new $c^{\omega}_{k^*}$, repeating the process until $|\text{fixed}_{\text{new}}| > 0$. Once $\text{fixed}_{\text{new}}$ is non-empty, we fix the identified weights $\{w'_1, w'_2, \ldots, w'_n\}$ to their corresponding cluster centre value $c^{\omega}_{k^*}$ and move them into W^{t+1}_{fixed}. We continue the process of identifying cluster centres and fixing weights to these centres until $|W^{t+1}_{\text{fixed}}| \geq Np_t$, at which point the iteration t is complete and the training stage of iteration $t+1$ begins. Our experiments found that a larger δ has less impact on task performance during early t iterations and so we use a decaying δ value schedule to maximise compression: $\delta^t = \delta(T - t + 1), t \in T$. We will show later that, with a small δ, over 75% of the weights can be fixed with $\omega = 1$ and over 95% of weights with $\omega \leq 2$.

3.2.6 Training Stage.

Despite the steps taken to minimise the impact of the clustering stage, without retraining, performance would suffer. To negate this, we perform gradient descent to adjust the remaining free weights W^t_{free}. This allows the weights to correct for any loss increase incurred after clustering where training aims to select values W^t_{free} that minimise the cross entropy loss $\mathcal{L}_{\text{cross-entropy}}$ whilst W_{fixed} remain unchanged.

3.2.7 Cosying up to Clusters.

Having the remaining W^t_{free} weights closer to the cluster centroids C post-training makes clustering less damaging to performance. We coerce this situation by adding to the retraining loss a regularisation term

$$\mathcal{L}_{\text{reg}} = \sum_{i \in W_{\text{free}}}^{N} \sum_{j}^{k} D^+_{\text{reg}}(w_i, c_j) p(c_j | w_i), \tag{3.5}$$

where $p(c_j|w_i) = \frac{e^{-D^+_{\text{reg}}(w_i, c_j)}}{\sum_l^k e^{-D^+_{\text{reg}}(w_i, c_l)}}$. The idea is to penalise the free-weights W^t_{free} in proportion to their distance to the closest cluster. Clusters that are unlikely to be weight w_i's nearest — and therefore final fixed value — do not contribute much to the penalisation term. We update the gradients of the cross-entropy training loss with the regularisation term:

$$\mathbf{w} \leftarrow \mathbf{w} - \eta \left(\nabla_{\mathbf{w}} \mathcal{L}_{\text{cross-entropy}} + \alpha \gamma \nabla_{\mathbf{w}} \mathcal{L}_{\text{reg}} \right),$$

Model	Data	Opt	LR	T	Batch size	γ	α
ResNet-18	ImageNet	Adam	2e-4	10	128	{0.05, 0.025, , 0.015, 0.01, 0.0075, 0.005}	{0.2, 0.4}
ResNet-34	ImageNet	Adam	2e-4	10	64	{0.05, 0.025, , 0.015, 0.01, 0.0075, 0.005}	{0.4}
ResNet-50	ImageNet	Adam	2e-4	10	64	{0.05, 0.025, , 0.015, 0.01, 0.0075, 0.005}	{0.4}
GoogLeNet	ImageNet	Adam	2e-4	10	64	{0.01, 0.0075, 0.015}	{0.4}
ResNet-18	CIFAR-10	Adam	3e-4	10	512	{0.01, 0.02, 0.03, 0.04, 0.05}	{0.0, 0.1, 0.2, 0.4, 0.8}
MobileNet	CIFAR-10	Adam	2e-4	10	512	{0.01, 0.02, 0.03, 0.04, 0.05}	{0.0, 0.1, 0.2, 0.4, 0.8}

TABLE 3.1: Full set of hyper-parameters explored for each model-dataset combination.

with α a hyper-parameter, and η the learning rate schedule. In our implementation we assign $\gamma = \frac{\mathcal{L}_{\text{reg}}}{\mathcal{L}_{\text{cross-entropy}}}$ and detach γ from the computational graph and treat it as a constant.

3.3 Experiment Details

We apply WFN to fully converged models trained on the CIFAR-10 and ImageNet datasets. Our pre-trained models are all publicly available with strong baseline accuracies[1]: Resnet-(18,34,50) [He et al., 2016] and, GoogLeNet [Chollet, 2017b]. We run ten weight-fixing iterations for three epochs, increasing the percentage of weights fixed until all weights are fixed to a cluster. In total, we train for 30 epochs per experiment using the Adam optimiser [Kingma and Ba, 2015] with a learning rate 2×10^{-5}. We use grid-search to explore hyper-parameter combinations using ResNet-18 models with the CIFAR-10 dataset and find that the regularisation weighting $\alpha = 0.4$ works well across all experiments reducing the need to further hyper-parameter tuning as we advance. The distance threshold δ gives the practitioner control over the compression-performance trade-off (see Figure 3.4), and so we report a range of values. We give a full breakdown of the parameters used across all experiments ran in Table 3.1.

3.4 Results

We begin by comparing WFN for a range of δ values against a diverse set of quantisation approaches that have comparable compression ratios (CR) in Table 3.2. The 3-bit quantisation methods we compare include: LSQ [Esser et al., 2020], LQ-Net [Zhang et al., 2018] and APoT [Yuhang Li, Xin Dong, 2020]. We additionally compare with the clustering-quantisation methods using the GoogLeNet model: Deep-k-Means [Wu et al., 2018] whose method is similar to ours, KQ [Yu et al., 2020], and GreBdec [Yu et al., 2017]. Whilst the results demonstrate WFN's lossless performance with SOTA CR, this is not the main motivation for the method. Instead, we are interested in how WFN can reduce the number of unique parameters in a network and corresponding weight-space entropy as well as the network representational cost, as defined in [Wu

[1]CIFAR-10 models : https://github.com/kuangliu/pytorch-cifar, ImageNet models: torchvision

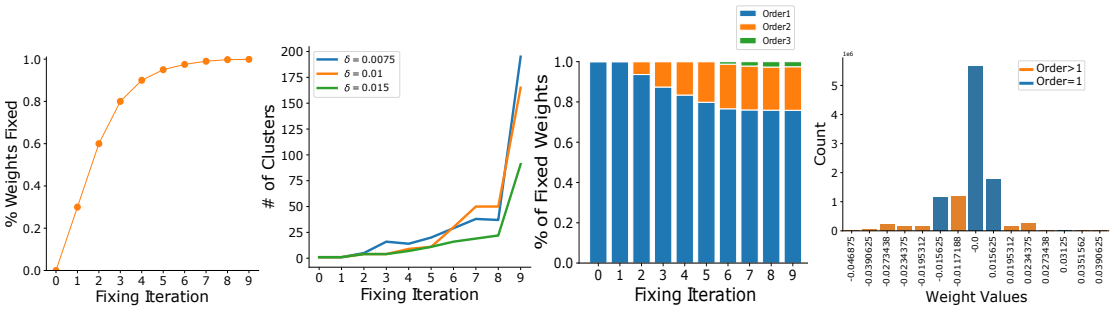


FIGURE 3.6: ImageNet experimental results: **Far left:** We increase the number of weights in the network that are fixed to cluster centres with each fixing iteration. **Middle left:** Decreasing the δ threshold increases the number of cluster centres, but only towards the last few fixing iterations, which helps keep the weight-space entropy down. **Middle right:** The majority of all weights are order 1 (powers-of-two), the increase in order is only needed for outlier weights in the final few fixing iterations. **Far right:** The weight distribution (top-15 most used show) is concentrated around just four values.

TABLE 3.2: A comparison of WFN against other quantisation and weight clustering approaches on the ImageNet dataset. The WFN pipeline is able to achieve higher compression ratios than the works compared whilst matching or improving upon baseline accuracies.

		Accuracy (%)					Accuracy (%)		
Model	**Method**	**Top-1**	**Top-5**	**CR**	**Model**	**Method**	**Top-1**	**Top-5**	**CR**
ResNet-18	Baseline	68.9	88.9	1.0	ResNet-34	Baseline	73.3	90.9	1.0
	LQ-Net	68.2	87.9	7.7		LQ-Net	71.9	90.2	8.6
	APoT	69.9	89.2	10.2		APoT	73.4	91.1	10.6
	LSQ	70.2^{+}	89.4^{+}	9.0*		LSQ	73.4^{+}	91.4^{+}	9.2*
	WFN $\delta = 0.0075$	70.3	89.1	10.2		WFN $\delta = 0.0075$	73.0	91.2	10.3
	WFN $\delta = 0.01$	69.7	89.2	12.3		WFN $\delta = 0.01$	72.6	91.0	11.1
	WFN $\delta = 0.015$	67.3	87.6	13.4		WFN $\delta = 0.015$	72.2	90.9	12.6
ResNet-50	Baseline	76.1	92.8	1.0	GoogLeNet	Baseline	69.7	89.6	1.0
	LQ-Net	74.2	91.6	5.9		Deep k-Means	69.4	89.7	3.0
	APoT	75.8	92.7	9.0		GreBdec	67.3	88.9	4.5
	LSQ	75.8^{+}	92.7^{+}	8.1*		KQ	69.2	-	5.8
	WFN $\delta = 0.0075$	76.0	92.7	9.5		WFN $\delta = 0.0075$	70.9	90.2	8.4
	WFN $\delta = 0.01$	75.4	92.5	9.8		WFN $\delta = 0.01$	70.5	90.0	8.4
	WFN $\delta = 0.015$	75.1	92.1	10.3		WFN $\delta = 0.015$	70.5	89.9	9.0

* Estimated from the LSQ paper model size comparison graph, we over-estimate to be as fair as possible.
$^{+}$ Open-source implementations have so far been unable to replicated the reported results: https://github.com/hustzxd/LSQuantization.

et al., 2018]. This metric has been tested and verified to linearly correlate with energy estimations deduced using the energy-estimation tool proposed in [Yang et al., 2017b]: $\text{Rep}(\mathcal{N}') = |W|N_w B_w$

Here, the representation cost is measured as the product of N_w, the number of operations weight w is involved in, B_w its bit-width and $|W|$, the number of unique weights in the network, respectively. This representation cost has a direct translation into data-movement costs since every weight ($|W|$) in the network will need to have its compressed representation (approx B_w if Huffman coding is used) moved from memory to where it is needed for compute N_w times.

TABLE 3.3: A full metric comparison of WFN Vs. APoT. Params refers to the total number of unique parameter values in the network. No BN-FL refers to the unique parameter count not including the first-last and batch-norm layers. WFN outperforms APoT even when we discount the advantage gained of taking on the challenge of quantising all layers. Model sizes are calculated using LZW compression.

Model	Method	Top-1	Entropy	Params	No BN-FL	$\text{Rep}_{\text{Mixed}}$	Model Size
ResNet-18	Baseline	68.9	23.3	10756029	10276369	1.000	46.8MB
	APoT (3bit)	69.9	5.77	1430	274	0.283	4.56MB
	WFN $\delta = 0.015$	67.3	2.72	90	81	0.005	3.5MB
	WFN $\delta = 0.01$	69.7	3.01	164	153	0.007	3.8MB
	WFN $\delta = 0.0075$	70.3	4.15	193	176	0.018	4.6MB
ResNet-34	Baseline	73.3	24.1	19014310	18551634	1.000	87.4MB
	APoT (3bit)	73.4	6.77	16748	389	0.296	8.23MB
	WFN $\delta = 0.015$	72.2	2.83	117	100	0.002	6.9MB
	WFN $\delta = 0.01$	72.6	3.48	164	130	0.002	7.9MB
	WFN $\delta = 0.0075$	73.0	3.87	233	187	0.004	8.5MB
ResNet-50*	Baseline	76.1	24.2	19915744	18255490	1.000	97.5MB
	WFN $\delta = 0.015$	75.1	3.55	125	102	0.002	9.3MB
	WFN $\delta = 0.01$	75.4	4.00	199	163	0.002	10.0MB
	WFN $\delta = 0.0075$	76.0	4.11	261	217	0.003	10.2MB

* The APoT model weights for ResNet-50 were not released so we are unable to conduct a comparison for this setting.

Due to the low weight-space entropy we achieve, we suggest Huffman encoding to represent the network weights (as is used by various accelerator designs [Moons and Verhelst, 2016a, Han et al., 2016a]). Given that the weight-representational bit-width will vary for each weight, we amend the original formulation to account for this, introducing

$$\text{Rep}_{\text{Mixed}}(\mathcal{N}') = \sum_{w_i \in W} N_{w_i} B_{w_i} \tag{3.6}$$

Here N_{w_i} is the number of times w_i is used in an inference computation, and B_{w_i} its Huffman-encoded representation bit-width of w_i.

The authors of the APoT have released the quantised model weights and code. We use the released model-saves[2] of this SOTA model to compare the entropy, representational cost, unique parameter count, model size and accuracy in Table 3.3. Our work outperforms APoT in weight-space entropy, unique parameter count and weight representational cost by a large margin. Taking the ResNet-18 experiments as an example, the reduction to just 164 weights compared with APoT's 9237 demonstrates the effectiveness of WFN. This huge reduction is partly due to our full-network quantisation (APoT, as aforementioned, does not quantise the first, last and batch-norm parameters). However, this does not tell the full story; even when we discount these advantages and look at weight subsets ignoring the first, last and batch-norm layers, WFN uses many times fewer parameters and half the weight-space entropy — see column 'No BN-FL' in Table 3.3. Finally, we examine how WFN achieves the reduced weight-space entropy in Figure 3.6. Here we see that not only do WFN networks have very few unique weights, but we also observe that the vast majority of all of the weights

[2] https://github.com/yhhhli/APoT_Quantization

			Full Network		No BN		No BN-FL	
Model	**Method**	**Top-1**	**Entropy**	**Param Count**	**Entropy**	**Param Count**	**Entropy**	**Param Count**
ResNet-18	Baseline	68.9	23.3	10756029	23.3	10748288	23.3	10276369
	APoT (3bit)	69.9	5.77	9237	5.76	1430	5.47	274
	WFN $\delta = 0.015$)	67.3	2.72	90	2.71	81	2.5	81
	WFN $\delta = 0.01$)	69.7	3.01	164	3.00	153	2.75	142
	WFN $\delta = 0.0075$)	70.3	4.15	193	4.13	176	3.98	162
ResNet-34	Baseline	73.3	24.1	19014310	24.1	18999320	24.10	18551634
	APoT (3bit)	73.4	6.77	16748	6.75	16474	6.62	389
	WFN $\delta = 0.015$)	72.2	2.83	117	2.81	100	2.68	100
	WFN $\delta = 0.01$)	72.6	3.48	164	3.47	132	3.35	130
	WFN $\delta = 0.0075$)	73.0	3.87	233	3.85	191	3.74	187
ResNet-50	Baseline	76.1	24.2	19915744	24.2	19872598	24.20	18255490
	WFN $\delta = 0.015$)	75.1	3.55	125	3.50	105	3.42	102
	WFN $\delta = 0.01$)	75.4	4.00	199	3.97	169	3.88	163
	WFN $\delta = 0.0075$)	76.0	4.11	261	4.09	223	4.00	217

TABLE 3.4: A full metric comparison of WFN Vs. APoT. We compare the unique parameter count and entropy of all parameters in the full network, as well as the same measures but not including the batch-norm layers (No BN) and the parameters not including the batch-norm and first and last layers (No BN-FL).

are a small handful of powers-of-two values (order 1). The other unique weights (outside the top 4) are low frequency and added only in the final fixing iterations.

3.5 Additional Analysis

3.5.1 Layerwise Breakdown

In Figure 3.7 we examine how the parameter count and layer-parameter entropy change with each layer for both the WFN and APoT approaches. We find both gains over the unquantised layers of APoT, but also that the entropy and parameter count in the convolutional layers (those quantised by APoT) are similar.

3.5.2 A Full Metric Comparison

In Table 3.4 we give the full metric breakdown comparing WFN to the state-of-the-art APoT work. We calculate the unique parameter count and entropy values for subsets of the weights. No BN corresponds to all weights other than those in the batch-norm layers, and No BN-FL is the set of weights not including the first-last and batch-norm layers. It is clear here that WFN outperforms APoT even when we discount the advantage gained of taking on the challenge of quantising all layers.

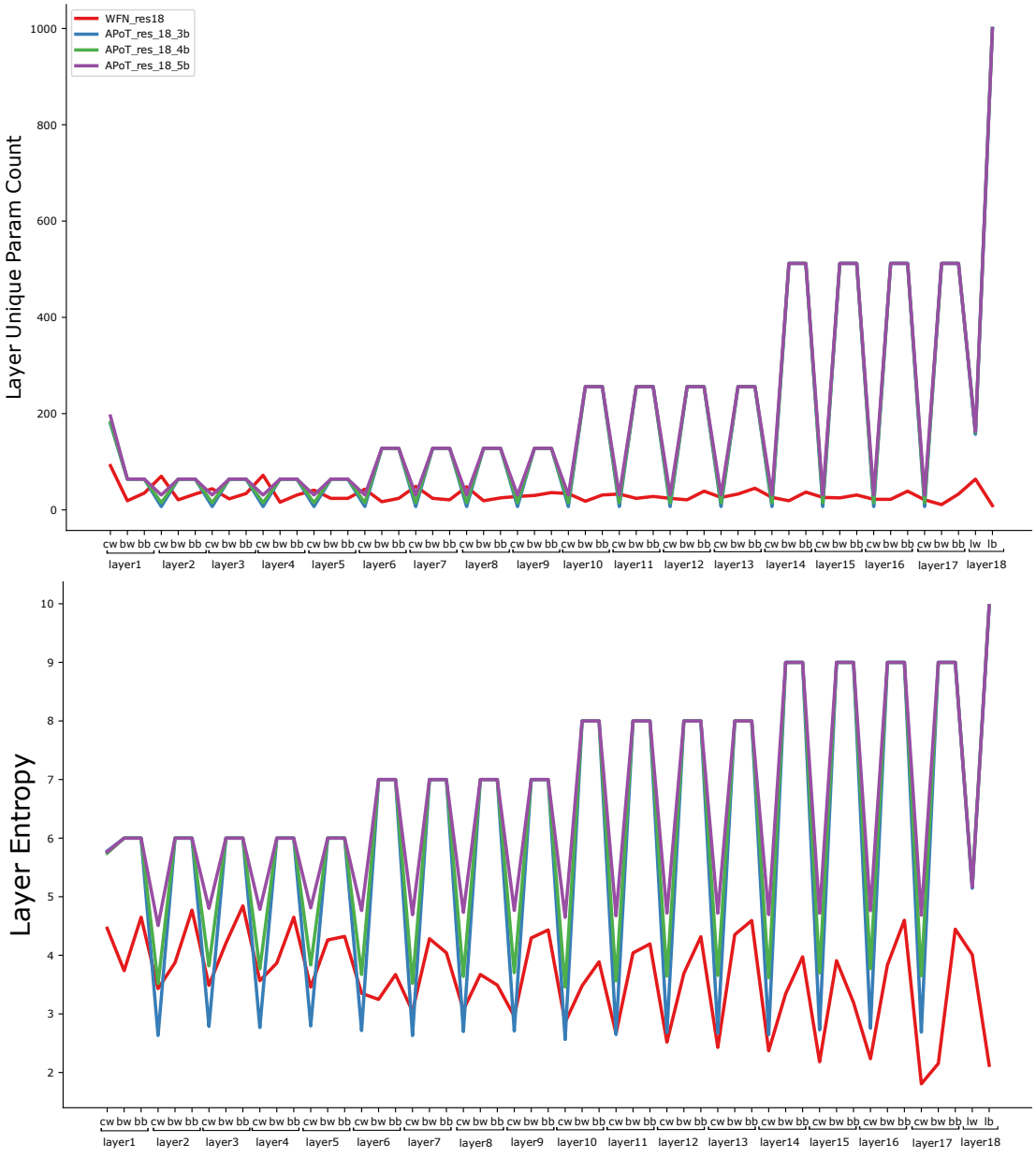


FIGURE 3.7: We compare WFN with a traditional quantisation set-up (APoT) with varying bit-widths applied to a ResNet18 model trained on the ImageNet dataset. The top chart shows the layerwise unique parameter count where WFN has consistently fewer unique parameters per layer.

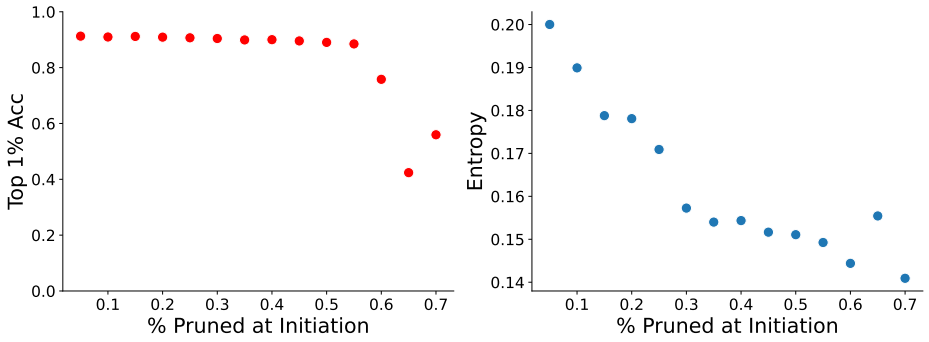


FIGURE 3.8: Here we show that unstructured pruning at initialisation up to 50% can be coupled with the WFN process without degradation of performance and can further reduce the weight-space entropy.

3.5.3 Pruning Experiments

To explore how WFN interacts with pruning we conduct a simple set of experiments. Instead of starting the WFN process with all weights un-fixed we randomly select p% of the weights to be pruned in each layer. We then run WFN as before starting with $p_t = p$, reducing the number of T iterations. The results, shown in Figure 3.8, are conducted with a ResNet-18 and Cifar-10 combination, painting a mixed picture. On the one hand, WFN and pruning at lower levels (< 50%) are well tolerated and provide two benefits, a lower weight-space entropy and few weight-fixing iterations. On the other hand, full-precision networks can tolerate much higher ranges of pruning so there it would seem that a certain amount of synergy between the two approaches is present but this is tempered compared to full precision networks.

It's important to note that WFN already has a form of pruning built-in with the δ_0 value balancing the emphasis on pruning over quantisation.

3.6 Conclusion

In this chapter we have presented WFN, a pipeline that can successfully compress whole neural networks. The WFN process produces hardware-friendly representations of networks using just a few unique weights without performance degradation. Our method couples a single network codebook with a focus on reducing the entropy of the weight-space along with the total number of unique weights in the network. The motivation is that this combination of outcomes will offer accelerator designers more scope for weight re-use and the ability to keep most/all weights close to computation to reduce the energy-hungry data movement costs. Additionally, we believe our findings and method offer avenues of further research in understanding the interaction between

network compressibility and generalisation, particularly when viewing deep learning through the minimal description length principle lens.

While the WFN presented in this chapter has shown promising results in compressing neural networks, there is an underlying assumption that small weights should move proportionately to their magnitudes. However, this assumption may not hold true for all individual weights, as some small weights could potentially move larger distances without significantly impacting accuracy. In the next chapter, we will explore an alternative approach that aims to address this limitation and in doing so makes a connection between compression, noise resilience and uncertainty estimations.

Chapter 4

Probabilistic Weight Fixing

4.1 An Introduction of Ideas

Our work in WFN demonstrated that it was possible to substantially reduce the number of unique weights and weight-space entropy whilst maintaining task performance.

In this chapter, we look further to refine the choices made in WFN. One fundamental assumption in WFN was that small weights should move proportionately to their current value. That is, small weights can move much smaller distances than larger weights. We found this empirically true when conducting the additive vs multiplicative noise experiments, which we show again in Figure 4.2. Taking the population-level perspective, we see that *on average* injecting noise relative to weight magnitude is better tolerated. However, this was from the perspective of entire weight populations and does not account for individual weight variability.

For example, say we have a network where weights w_i and w_j are both 0.23, and we would like to apply the WFN algorithm. Both weights would be considered equally viable to be moved to 0.25 at this stage since their relative magnitudes are the same.

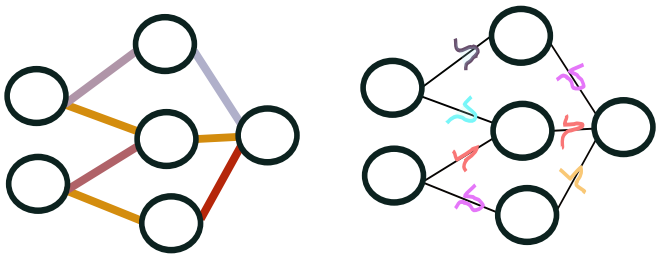


FIGURE 4.1: The weight values for any given forward pass through a Bayesian network are drawn from a learned distribution (right) rather than being point estimates (left).

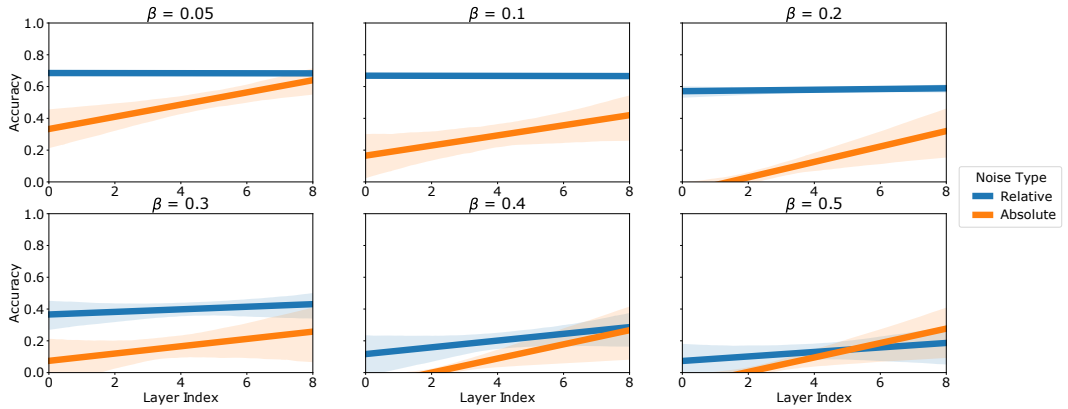


FIGURE 4.2: In WFN, we explored adding relative vs absolute noise to each of the layers (x-axis). The layer index indicates which layer was selected to have noise added. Each layer index is a separate experiment with the 95% confidence intervals shaded. It's clear here that the relative noise is much better tolerated at a population level.

However, this might not be the case; location might also matter here. Perhaps w_i is finely tuned to its neighbourhood values such that even a small perturbation would result in a change to downstream activation values and even change the classification class, but w_j has much more flexibility and can be increased or decreased to a greater extent. The core idea is that relative magnitude, although a good proxy for determining how much a weight can be moved, ignores neighbourhood and positioning.

Could some weights with small values move large distances without having any meaningful impact accuracy, and some larger weights be extremely sensitive to even tiny perturbation shifts? This question motivates this Chapter and a new technique which we call *Probabilistic Weight Fixing Networks* (PWFN). Rather than enforcing small relative distance movements irrespective of weight position we look to training and using Bayesian neural networks (BNN) [Neal, 2012, Mackay, 1992, Gal et al., 2016] and using the uncertainty and resilience to noise perturbations found in the optimisation process to determine which weights to move where. We will later define BNNs formally, but for now, it serves us to give a simple overview.

4.1.1 High-level BNN

The fundamental concept of BNNs is that each weight in the network is derived from a probability distribution, rather than being assigned a fixed value. This means that during prediction, we randomly select a single value from the distribution of each weight to use in the network. As a result, given the same input, different forward passes through the network could lead to variations in the weights' values, and potentially, in the predicted output. Each weight in the network is characterised by a set of parameters

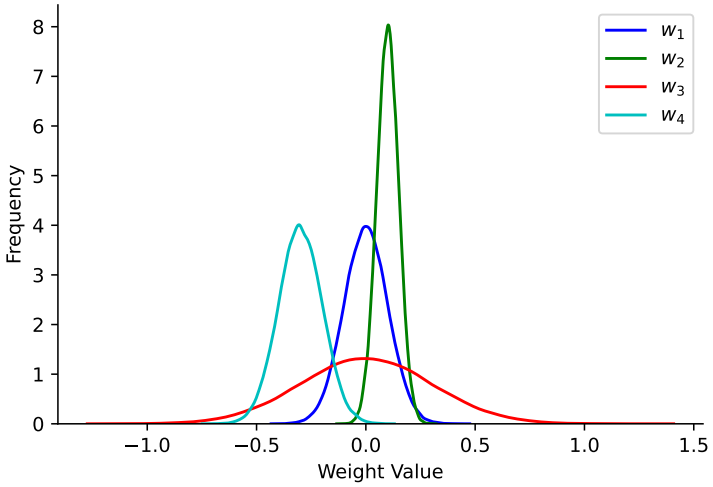


FIGURE 4.3: Here we show a set of four distributions — one for each of the four weights — chosen randomly. In the forward pass, we will sample one sample per weight from these distributions, according to the probability density function.

that define its distribution, which are sampled during the inference process. These parameters, rather than the weights themselves, are what we learn during training.

In this chapter, we explore how to learn and utilise these distribution parameters to assist in the process of clustering and quantisation of the weights. By having a distinct set of distribution parameters for each weight, we can move beyond using a single, population-based threshold for quantisation.

Before diving further into the details, let's look at a simple example to motivate the idea. Imagine that we have a linear model of the form:

$$y = \sum_{i=1}^{4} w_i x_i, \tag{4.1}$$

where, in the usual way, w_i corresponds to a weight, x_i to an input, and y the output of the model. Rather than attempting to learn w_i's directly, we assume that each w_i comes from a Gaussian distribution $w_i \sim \mathcal{N}(\mu_i, \sigma_i)$ and the task of learning is one of finding good parameters μ and σ for each of the weights to minimise performance loss. For now, let's assume we have such a learning process and let's look at a learned possible outcome post-training in Figure 4.4 (left). With each of the four weights' distributions is colour-coded we can ask ourselves what information we learn from these parametrisations that might be useful for quantisation/clustering.

If we assume that we have learned these distributions of weights during the training process whilst still maintaining task performance, then we have learned a lot about how

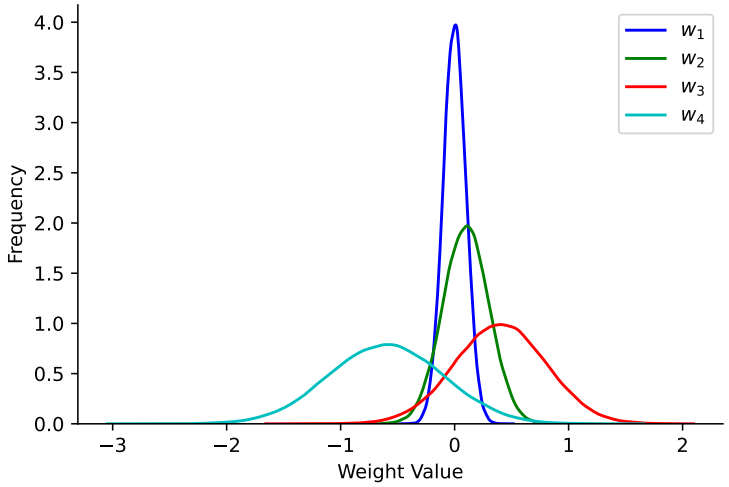


FIGURE 4.4: If the previous assumption made in WFN, that large magnitude weights can move larger distances than smaller weights; we would expect the weights' distributions to learn larger σ values for larger magnitude weights which we show one example of here on the right.

much we can move each weight value whilst still keeping the weight within the scope of values that have been sampled during training. w_2, for example, is very sensitive to noise perturbations since, during training, it has learned to reduce the σ_2 value to be small. Having smaller σ values indicates that we shouldn't move this weight too far during clustering — its *quantisation error* should be kept small. Conversely, the distribution learned for w_3 has a much larger tail ($\sigma_3 \gg \sigma_2$), indicating that w_3 has been sampled at a larger range of values in training without affecting performance and can therefore be moved more liberally during clustering.

Recall in WFN, we worked assuming that large weights are better tolerated than smaller weights to move more considerable distances. What would a realisation of this hypothesis look like if we learned the Gaussian distributions? Figure 4.4 shows one such example that we would expect to see if this were the case, where the sigma values are proportional to the distribution means.

4.1.2 The Two Problems to Solve

Within this chapter, we have two problems to solve - how to model a neural network with the weights stored as distributions so that the parameters both inform clustering whilst maintaining performance. And, how best to use the information within the resulting network to perform clustering to the maximal degree.

4.1.2.1 Problem One: Modelling

The primary challenge in BNNs is effectively modelling and learning the distributions for each weight in the network. To address this, we draw on established BNN literature, introducing some modifications to the standard BNN training setup. These modifications are designed to mitigate the intractability of the full Bayesian approach, which involves solving for all possible configurations of the weights and then weighting them by their likelihood. The full Bayesian approach becomes intractable due to the high-dimensional integral required to compute the posterior distribution over the weights, which grows exponentially with the number of weights in the network.

One popular approach is variational inference, which involves introducing a variational distribution over the weights and optimising it to minimise the Kullback-Leibler (KL) divergence between the variational distribution and the true posterior. The KL divergence measures the difference between two probability distributions, and minimising it ensures that the variational distribution closely approximates the true posterior. By restricting the variational distribution to a simpler family of distributions, such as Gaussian distributions, the optimisation problem becomes tractable and can be solved using standard gradient-based methods.

A fundamental challenge with the Gaussian distribution variational approach is determining the appropriate prior for the standard deviation (σ) [Gal et al., 2016, Maddox et al., 2019]. If no prior is given and only a performance loss is used, the optimisation process would be incentivised to collapse σ values to zero and only use the μ values. This is because reducing the σ values to zero eliminates the uncertainty in the weights and simplifies the optimisation problem. So a prior is needed to prevent this collapse, but it is not intuitively obvious what such a prior should be.

Another interrelated issue is initialising the parameters of these distributions to ensure convergence. This is crucial in BNNs, where the forward pass signal is derived from multiple parameters, thereby diluting the signal in the backward pass. This weakened signal necessitates more training iterations, making a robust initialisation scheme critical to complete training within a reasonable timeframe. A robust initialisation scheme should provide a good starting point for the optimisation process, helping the model converge faster and avoid getting stuck in suboptimal solutions.

Due to these challenges, scaling BNNs to complex datasets like ImageNet, particularly using variational inference, has been difficult. The most effective method to approximate the posterior distribution of weights in such complex scenarios has been through observing the Stochastic Gradient Descent (SGD) trajectory [Maddox et al., 2019].

In this chapter, we propose that a network's resilience to noise is a key indicator of its downstream performance. We argue that by incorporating noise resilience as a regularisation term in the optimisation objective, we can effectively guide the learning of the weight distributions in BNNs. This perspective allows us to regularise BNNs effectively, encouraging the model to learn weight distributions that are robust to noise perturbations. By promoting noise resilience, our approach facilitates the training of BNNs with more complex dataset-model combinations using the variational Gaussian distribution, enabling their application to a wider range of real-world problems.

4.1.2.2 Problem Two: Utilising What We Learn

Once we have addressed the challenges in modelling and training BNNs, resulting in a trained network with each weight represented by a distribution, the next problem is to determine how to effectively utilise this information in the context of clustering and quantisation.

A key question arises: Can the uncertainty learned during the training of a BNN be leveraged to inform the quantisation stage? In other words, can we use the information encoded in the weight distributions to guide the process of clustering and quantising the weights? This is a crucial consideration, as the uncertainty captured by the weight distributions may provide valuable insights into the importance and sensitivity of different weights in the network. In the second part of this chapter, we will explore methods to integrate the learned BNNs into a clustering/quantisation algorithm. We hypothesise that the uncertainty learned during training can serve as a valuable source of information for clustering. By incorporating this uncertainty into the clustering process, we aim to develop more efficient and effective quantisation techniques that are tailored to the specific characteristics of the trained BNN.

Before delving into the details of our Probabilistic Weight Fixing Networks (PWFN) algorithm, let's take a moment to review the broader landscape of BNNs. We will discuss the key motivations driving research in this field and highlight some of the open challenges that our work aims to address. This background information will provide context for our contributions and help situate our approach within the existing body of knowledge.

4.2 Background on Bayesian Neural Networks

Bayesian Neural Networks (BNNs) introduce a fundamental change in weight modelling compared to traditional neural networks. Instead of assigning fixed values to

weights, BNNs consider a probability distribution over each weight (see Figure 4.1). Ideally, Bayesian inference would involve computing the integral

$$P(\boldsymbol{y}|x, D) = \int_{w} P(\boldsymbol{y}|x, \boldsymbol{w}) P(\boldsymbol{w}|D)\, dw,$$

where predictions are averaged across all possible weights $\boldsymbol{w}$. This concept, although theoretically appealing, is computationally infeasible for complex networks, leading to the exploration of alternative methods for approximating the Bayesian posterior.

4.2.1 Origins of Bayesian Neural Networks

One of the earliest and most influential works in this domain was by David MacKay in the early 1990s, notably his 1992 paper, "A Practical Bayesian Framework for Backprop Networks" [Mackay, 1992]. This work laid the foundation for applying Bayesian principles to neural network weights, thus addressing overfitting by incorporating prior knowledge into the learning process.

Radford Neal's 1995 doctoral thesis, "Bayesian Learning for Neural Networks," [Neal, 2012] further advanced the field. Neal proposed viewing large neural networks as Gaussian processes, a radical idea that reframed neural networks as probabilistic models capable of quantifying uncertainty.

4.2.2 Challenges and Conceptual Shifts

Despite these initial advancements, the development of BNNs faced significant obstacles. The primary challenge was computational; the Bayesian methods, especially those involving probabilistic weight interpretation, demanded substantial computational resources. This limitation hindered their application to larger networks and datasets.

Additionally, the transition from perceiving weights as fixed entities to probabilistic distributions necessitated a substantial shift in understanding and training neural networks. This change presented both technical and theoretical challenges, requiring a deep integration of Bayesian statistics and neural network architecture [Jordan and Mitchell, 2015].

4.2.3 Advantages of Bayesian Approaches

The shift towards Bayesian methodologies in neural networks is motivated by several key advantages of BNNs over traditional models.

Firstly, BNNs inherently provide a measure of uncertainty in their predictions, which is crucial in various applications such as medical diagnosis and financial forecasting. In these domains, understanding the confidence level of predictions is as important as the predictions themselves [Gal et al., 2016].

Secondly, BNNs naturally mitigate overfitting. Traditional neural networks often suffer from overfitting, performing well on training data but poorly on new data. BNNs, by integrating prior information and considering a range of weight distributions, generalise better to unseen data [Welling and Teh, 2011].

Additionally, BNNs offer a more principled approach to model complexity. By allowing the data itself, under probabilistic principles, to guide the determination of model complexity, BNNs can lead to more optimally sized models that neither underfit nor overfit the data [Graves, 2011].

4.2.4 Contemporary Research Trends

Recent innovations in BNNs include the development of subnetworks within the model that are Bayesian in nature [Sharma et al., 2023, Daxberger et al., 2021], the application of various approximation techniques [Kristiadi et al., 2021, Maddox et al., 2019], and the use of Gaussian variational relaxations, as demonstrated by Blundell et al. [Blundell et al., 2015]. Their 'Bayes-by-backprop' (BBP) method, in particular, has expanded the scope of BNNs to modern deep learning architectures using a re-parametrization trick, enabling back-propagation optimisation and uncertainty quantification in large-scale networks. Our proposed method, PWFN builds upon these advancements and is an extension of the variational approach - which we'll now look to explore more with.

4.2.5 Variational BNNs

Recall that Bayesian inference would involve computing the integral $P(\boldsymbol{y}|x, D) = \int_{\boldsymbol{w}} P(\boldsymbol{y}|x, \boldsymbol{w})P(\boldsymbol{w}|D)\, dw$. The key term here that is difficult to compute is the posterior distribution of the weights given our data $P(\boldsymbol{w}|D)$. A variational approximation of this posterior looks to find parameters θ such that variational distribution $Q(w|\theta)$ minimises the Kullback-Leibler (KL) divergence defined as $D_{KL}(Q||P) = \int Q(x) \log \frac{Q(x)}{P(x)} dx$ which can be reformulated [**?**] as:

$$
\begin{aligned}
D_{KL}(Q||P) &= \int Q(\boldsymbol{w}|\theta) \log \frac{Q(\boldsymbol{w}|\theta)}{P(\boldsymbol{w}|\boldsymbol{D})} dw && \text{Definition of KL divergence} \\
&= \int Q(\boldsymbol{w}|\theta) \log \frac{Q(\boldsymbol{w}|\theta)P(\boldsymbol{D})}{P(\boldsymbol{w},\boldsymbol{D})} dw && \text{Bayes theorem} \\
&= \int Q(\boldsymbol{w}|\theta) \log \frac{Q(\boldsymbol{w}|\theta)}{P(\boldsymbol{w},\boldsymbol{D})} dw + \int Q(\boldsymbol{w}|\theta) \log P(\boldsymbol{D}) dw && \log ab = \log a + \log b \\
&= \int Q(\boldsymbol{w}|\theta) \log \frac{Q(\boldsymbol{w}|\theta)}{P(\boldsymbol{w},\boldsymbol{D})} dw + \log P(\boldsymbol{D}) && \int Q(\boldsymbol{w}|\theta) dw = 1 \\
\log P(\boldsymbol{D}) &= D_{KL}(Q||P) - \int Q(\boldsymbol{w}|\theta) \log \frac{Q(\boldsymbol{w}|\theta)}{P(\boldsymbol{w},\boldsymbol{D})} dw && \text{Switch sides and negate} \\
\log P(\boldsymbol{D}) &= D_{KL}(Q||P) + \mathcal{L}(Q) && \mathcal{L}(Q) = -\int Q(\boldsymbol{w}|\theta) \log \frac{Q(\boldsymbol{w}|\theta)}{P(\boldsymbol{w},\boldsymbol{D})} dw.
\end{aligned}
$$

We know that the probability of our data $P(D)$ is fixed and since the KL term is always positive, $D_{KL}(Q||P) \geq 0$, if we maximise $\mathcal{L}(Q)$ we will be simultaneously decreasing the KL divergence term. This term, $\mathcal{L}(Q)$, is known both as the variational free energy and also the evidence lower bound (ELBO).

Looking closely at $\mathcal{L}(Q)$:

$$
\begin{aligned}
\mathcal{L}(Q) &= -\int Q(\boldsymbol{w}|\theta) \log \frac{Q(\boldsymbol{w}|\theta)}{P(\boldsymbol{w},\boldsymbol{D})} dw \\
&= -\int Q(\boldsymbol{w}|\theta) \log \frac{Q(\boldsymbol{w}|\theta)}{P(\boldsymbol{D}|\boldsymbol{w})P(\boldsymbol{w})} dw \\
&= -\int Q(\boldsymbol{w}|\theta) \log Q(\boldsymbol{w}|\theta) dw + \int Q(\boldsymbol{w}|\theta) \log P(\boldsymbol{D}|\boldsymbol{w})P(\boldsymbol{w}) dw \\
&= H(Q(\boldsymbol{w}|\theta)) + \int Q(\boldsymbol{w}|\theta) \log P(\boldsymbol{D}|\boldsymbol{w})P(\boldsymbol{w}) dw \\
&= H(Q(\boldsymbol{w}|\theta)) + \int Q(\boldsymbol{w}|\theta) \log P(\boldsymbol{D}|\boldsymbol{w}) dw + \int Q(\boldsymbol{w}|\theta) \log P(\boldsymbol{w}) dw,
\end{aligned}
$$

we can check in with an explanation for each of the three components we would like to maximise to minimise the KL divergence $D_{KL}(Q||P)$. The first, $H(Q(w))$, corresponds to the entropy of the weights w coming out of the Q family distribution. To maximise this term, we could look to distributions which maximise entropy and then encourage solutions (parameter settings) in the optimisation process which also increase the entropy. The second term $\int Q(\boldsymbol{w}|\theta) \log P(\boldsymbol{D}|\boldsymbol{w})$ refers to the weighted expectation of

true model likelihood over the variational Q. Finally, we have $\int Q(\boldsymbol{w}|\theta) \log P(\boldsymbol{w})$ which is the cross-entropy of $Q(w|\theta)$ and $P(w)$.

Even with the selection of a simple variational distribution $Q(\boldsymbol{w}|\theta)$, the minimisation of the variational free energy has no tractable closed-form solution for problems of significant complexity. Instead, sampling approximation methods (such as Metropolis-Hastings) or gradient-based methods, like Bayes By Backprop (BBP), can be used. However, even with these relaxations, modelling full deep learning networks as a set of Gaussian distributions and solving complex problems remains challenging. For example, in the BBP paper [Blundell et al., 2015], experiments were limited to MNIST classification and other toy problems. Despite these challenges, variational inference provides a principled framework for approximating the intractable posterior distribution in BNNs. By introducing a variational distribution and optimising the ELBO, we can effectively learn the parameters of the variational distribution and obtain an approximation to the true posterior. This enables us to perform efficient inference and make predictions with uncertainty estimates. In the following sections, we will explore how our proposed method, PWFN, builds upon the variational inference framework and addresses some of the limitations of existing approaches.

4.3 Probabilistic Weight Fixing Networks

We are now ready to begin to introduce PWFNs. In doing so, our core idea is to integrate three fields, stochastic/Bayesian neural networks, quantisation, and accelerator designs. We will use the curious juxtaposition gained from quantisation results – that networks that can tolerate noise are *better* than those that can't – to train BNNs even for the more complex model-dataset combinations. We will then use the information stored in the parameters of the weights distributions to guide quantisation/clusterings towards configurations that offer the opportunities to maximise weight reuse - a core consideration and driver of accelerator designs in reducing the overall energy costs.

4.3.1 High Level Overview

In PWFN, we follow T fixing iterations each of which combines a training and a clustering stage in order to reach an outcome of a highly compressed/quantised network with a single whole-network codebook. In our approach, each weight w_i in the set $\mathbf{w}$ is modelled as a sample from a Gaussian distribution, represented as $\mathcal{N}(\mu_i, \sigma_i)$. These distributions are characterised by learnable parameters: means $\boldsymbol{\mu} = (\mu_1, \ldots, \mu_N)$ and standard deviations $\boldsymbol{\sigma} = (\sigma_1, \ldots, \sigma_N)$. During the training phase, we employ Bayesian by Backpropagation (BBP) to optimise these parameters. The objective is

twofold: firstly, to minimise the loss associated with task performance, and secondly, to refine the weight distributions. This refinement aims to precisely quantify the extent of noise that can be introduced to each weight w_i without adversely impacting the model's performance. Both $\boldsymbol{\mu}$ and $\boldsymbol{\sigma}$ are trained with an additional regularisation term that encourages larger values of $\boldsymbol{\sigma}$ to counter the model reverting to the point estimates with $\sigma_i = 0$ to minimise the classification loss. During the clustering stage, we look to use this information to move the μ_i values to one of a handful of cluster centers. We favour the cluster centers to be hardware multiplication-friendly powers-of-two, or additive-powers-of-two. After T iterations of training and clustering, each of the weights' distributions in the networks will have their $\boldsymbol{\mu}$ values centered on one of the k clusters in the codebook $\boldsymbol{c}$.

After the T fixing iterations there are two options depending on the downstream usage of the network: either the network can be converted into point estimates and the weights set to the exact μ values giving us a quantised network. Or, we can use the extra information given to us by modelling each weight as a distribution as a way of quantifying uncertainty of a particular prediction. If, after multiple samples of $\boldsymbol{w}$, a model changes its prediction for a fixed input, this tells us that there is uncertainty in these predictions – with this information being useful for practical settings.

Let us now dive into the training phase of PWFN.

4.3.2 PWFN Training

Consider a network parameterized by N weights $\boldsymbol{w} = \{w_1, ..., w_N\}$. In PWFN, each weight w_i is not a single value but is instead drawn from a distribution $w_i \sim \mathcal{N}(\mu_i, \sigma_i)$, and instead of learning the w_i directly, the learning process optimizes each μ_i and σ_i. In a forward pass, during training, we sample weight values w_i according to its distribution:

$$w_i = \mu_i + \sigma_i \epsilon, \ \epsilon \sim \mathcal{N}(0, 1). \tag{4.2}$$

The forward pass is stochastic under fixed $\boldsymbol{\mu}, \boldsymbol{\sigma}$. If trained correctly, the σ_i values give us information about the amount of noise a particular weight w_i can handle without affecting performance. Said another way, if we can find a configuration $\boldsymbol{w} = (\boldsymbol{\mu}, \boldsymbol{\sigma})$ which maintains task performance despite the randomness introduced by the σ_i parameters, then we will know which of the corresponding weights can be moved and to what degree. In PWFN, we train $\boldsymbol{\mu}$, $\boldsymbol{\sigma}$ following the BBP optimisation process [Blundell et al., 2015] with some changes both in terms of initalisation and the priors on $\boldsymbol{\mu}$ and $\boldsymbol{\sigma}$.

4.3.3 Large σ constraint for w.

Given the usual cross-entropy or other performance loss, there is a clear direction of travel during gradient descent towards having small σ_i values and less uncertainty in the network parameters. If σ_i's are kept small (close to zero) then the optimisation process need only find a good placement for μ_i's to minimise the loss. A prior on the distribution of weights is therefore needed to prevent the $\boldsymbol{\sigma} = 0$ point estimate solution being found which would leave us with no weight movement information.

In the original BBP set-up, the authors aimed to prevent vanishing variance by regularising the distribution of weights according to a prior distribution composed of a mixture of zero-mean Gaussian densities with different variances. The parameters of this prior were determined through an exhaustive search. The motivation for this approach was twofold: firstly, the empirical Bayes method did not perform well because the network tended to favour updating these parameters over the posterior (since there are fewer parameters to update); secondly, there was a connection to the successful spike-and-slab prior [**?**], where values are concentrated around 0 (the *slab*) or another value known as the *spike*, thus favouring sparsity.

In contrast, we hypothesise that a *good* network can handle the most noise injection whilst maintaining performance. Such networks are likely more compressible, as they have been trained to tolerate changes to their weight values without performance degradation during training.

We attempt this by making our $\boldsymbol{\sigma}$ values to be large. Networks with large $\boldsymbol{\sigma}$ have, probabilistically, more noise added to the $\boldsymbol{\mu}$ values during training and so have to learn to have robust performance under such circumstances. We note that this acts as a push-pull relationship with the performance loss, which favours low $\boldsymbol{\sigma}$ values. The motivation is that, much like L_1 norms enforcing sparsity, this formulation will train the network to produce a large σ_i for noise-resilient parameter w_i, whilst maintaining a noise-sensitive weight w_j to have a small σ_j despite the prior pull. The regularised loss function for training the training phases of the algorithm is:

$$-\log P(\mathcal{D}|\boldsymbol{\mu}, \boldsymbol{\sigma}) + \alpha \mathcal{L}_{\text{REG}}(\boldsymbol{\sigma}), \tag{4.3}$$

where the regularisation term is:

$$\mathcal{L}_{\text{REG}}(\boldsymbol{\sigma}) = \sum_{i=1}^{N} \mathcal{L}(\sigma_i) = -\sum_{i=1}^{N} (\sigma_i - S)\Theta(S - \sigma_i), \tag{4.4}$$

with $\Theta(x) = 1$ for $x \geq 0$ and 0 otherwise. The Θ function prevents the optimization from finding a network with a subset of $\boldsymbol{\sigma}$ with infinitely large values and dominating the cross entropy term. S is thus a cutoff on how large the values in $\boldsymbol{\sigma}$ can be. α is a

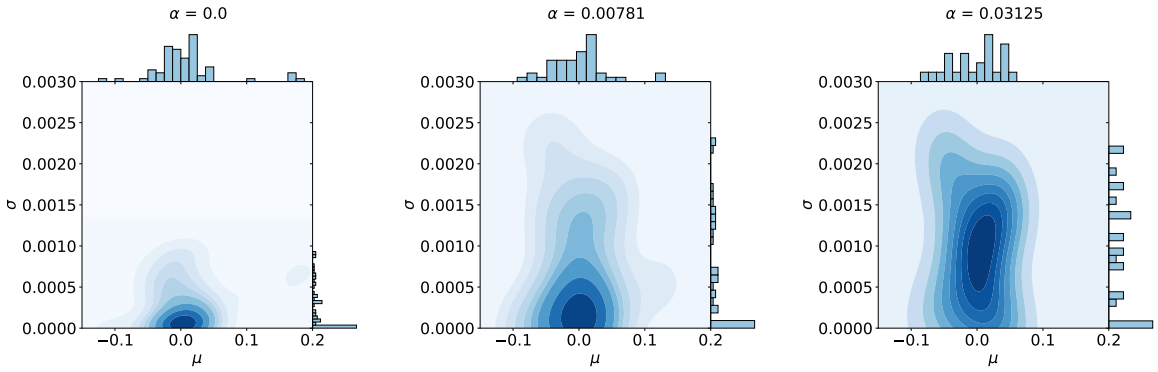


FIGURE 4.5: The regularisation term acts to stop the σ uncertainty values from collapsing to zero. This experiment is run using the CIFAR10 dataset with ResNet-18, stopping after 30 epochs.

hyperparameter controlling the formation of a noise-resilient network where the *majority* of the weights can receive noise injection without hurting performance, not just a few. In Figure 4.5 we illustrate the effect on the distribution of σ under different α values for a ResNet-18 trained on the CIFAR-10 dataset. As we increase α the σ values no longer collapse to zero giving us information for downstream clustering.

4.3.4 Initialization using Relative Distance from Powers-of-two.

For each weight w_i we need to specify its prior distribution so as to derive the posterior using Bayesian updating. We assume that the posterior distribution is Gaussian with a diagonal covariance matrix: $P(w_i; \mu_i, \sigma_i)$ whose parameters μ_i, σ_i are trained using BBP. To initialise the prior distributions for μ_i and σ_i we set:

$$P^0(\mu) = \prod_i P^0(\mu_i) \tag{4.5}$$

where

$$P^0(\mu_i) \propto \delta_{\mu_i, w_i} \tag{4.6}$$

,

for the pre-trained weight value w_i. For a Gaussian posterior we would typically require an unknown σ to be drawn from a Gamma conjugate prior distribution. Instead, we set σ_i to be a known function of the μ_i at initalisation. In our previous chapter with WFN[Subia-Waud and Dasmahapatra, 2022] relative distances to the preferred powers of two values for the weight was used to determine weight movement. To favour anchoring weights at powers of two, we set the standard deviations to be smallest (2^{-30})

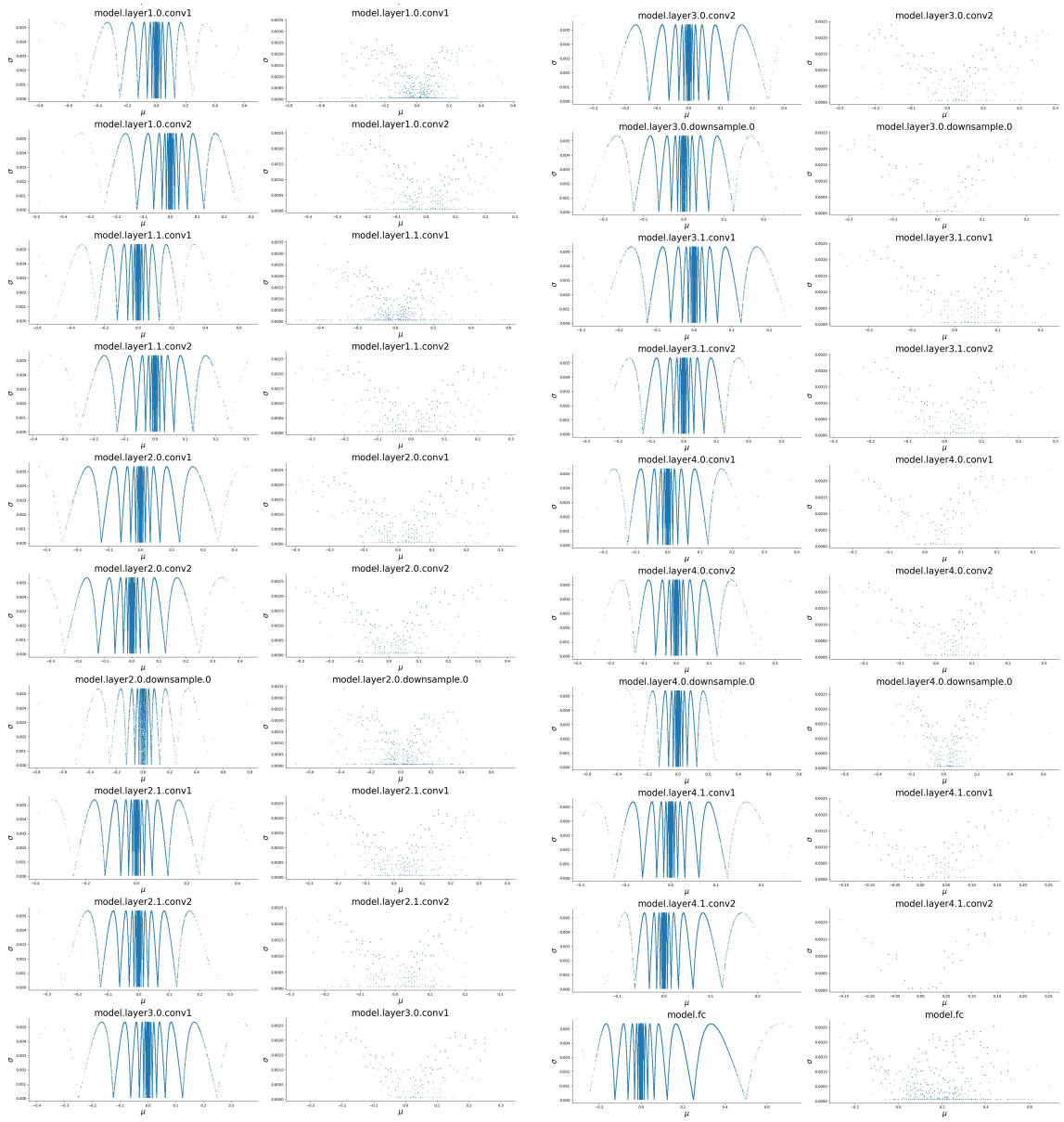


FIGURE 4.6: Here we compare the μ vs σ values for all weights in a given layer at initalisation (left) and after PWFN convergence and clustering (right).

at either edge of each interval between the nearest integer powers of two (remembering that $2^{x_i} \leq \mu_i \leq 2^{x_i+1}$) for integer x_i, and largest at the midpoint of the interval. We introduce a parabolic function $\sigma_i(\mu_i)$ as a product of relative distances of each pre-trained weight value (μ_i) to the nearest lower and upper powers of two:

$$\sigma_i(\mu_i) = (0.05)^2 \left(\frac{|2^{x_i} - \mu_i|}{|2^{x_i}|} \right) \left(\frac{|\mu_i - 2^{x_i+1}|}{|2^{x_i+1}|} \right) \tag{4.7}$$

We show a set of initialised μ verses σ values and the converged values post-training for a ResNet-18 model trained for CIFAR10 classification in Figure 4.6.

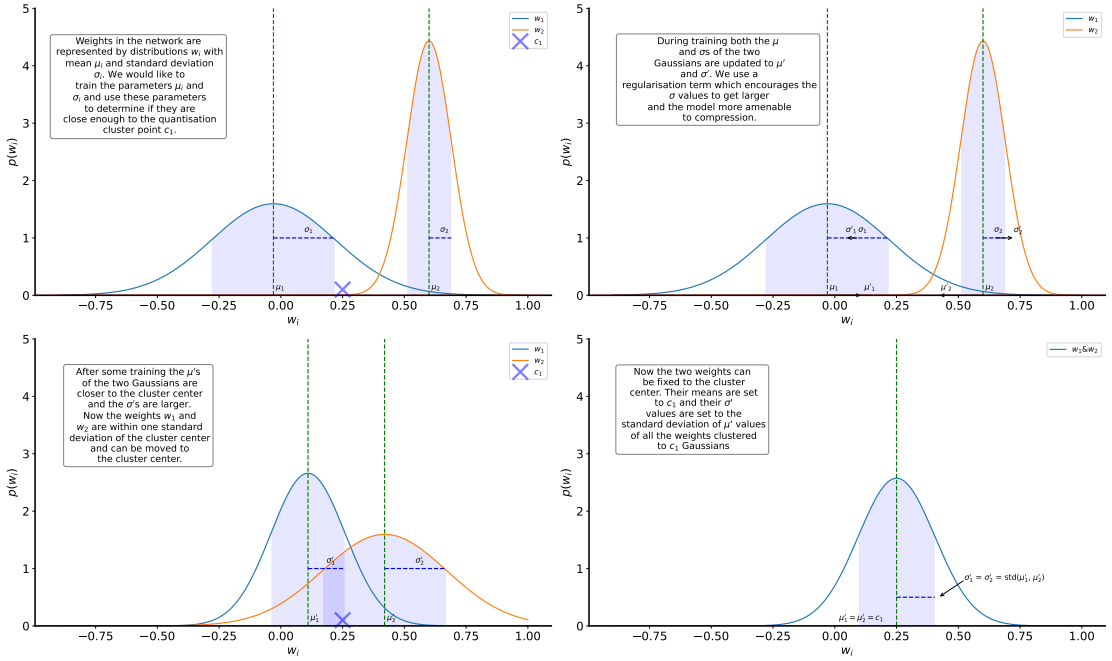


FIGURE 4.7: An overview of the PWFN process.

4.3.5 PWFN Clustering.

In Figure 4.7 we show a schematic of the clustering stage in which we use the information garnered from the weights' distribution parameters to identify cluster centers and their assignment. PWFN clustering is a two-step method running for $t = 1, \ldots, T$ iterations. At each step we set a fraction p_t of the weights to be fixed, so that $|W^t_{\text{fixed}}| = Np_t$. The remaining weights at iteration stage t are trainable and called W^t_{free}. We follow the scheme first proposed in [Subia-Waud and Dasmahapatra, 2022] in setting p_t (Figure 4.8, left). All of the weights w_i that are assigned to W^t_{fixed} will have their μ_i values fixed to one of the set of cluster centers. At the last iteration, $|W^T_{\text{free}}| = 0$ and $p_T = 1$, as all weights have been fixed to their allocated cluster centroids.

We next introduce how a cluster center c_k is defined and how the mapping $\mu_i \mapsto c_k \in \boldsymbol{c}$ is performed. Let

$$R = \{-\frac{1}{2^b}, \ldots, -\frac{1}{2^{j+1}}, -\frac{1}{2^j}, 0, \frac{1}{2^j}, \frac{1}{2^{j+1}}, \ldots \frac{1}{2^b}\} \tag{4.8}$$

be the set of all powers-of-two up to a precision b. For a weight to be a desired additive power of two, a sum over at most ω elements of R is defined to be a cluster center of order ω. Formally, for $\mathcal{P}(R)$ the power set of R,

$$\boldsymbol{c}^\omega = \{\sum_{i \in r} i \mid r \in \mathcal{P}(R) \wedge |r| \leq \omega\}. \tag{4.9}$$

PWFN begins with order $\omega = 1$, the powers-of-two up to precision b as the proposal cluster set $\boldsymbol{c}^{\omega}$. Next, for each weight $w_i = (\mu_i, \sigma_i)$ in the network, the value of σ_i is used to determine how far away they are from each of the cluster centers using:

$$D_{\text{prob}}(w_i, c_j) = \frac{|\mu_i - c_j|}{\sigma_i}. \tag{4.10}$$

Interpret this Mahalanobis distance as: "how many sigmas (standard deviations) away is cluster $c_j \in \boldsymbol{c}^{\omega}$ from weight w_i". At iteration stage t, for each free weight we define

$$c_*^{\omega}(i) = \min_{c \in \boldsymbol{c}^{\omega}} D_{\text{prob}}(w_i, c) \tag{4.11}$$

as the cluster center that is the fewest sigmas away from $w_i \in W_{\text{free}}^t$. We denote by n_k^{ω} the number of weights with the smallest D_{prob} to cluster c_k^{ω}, *i.e.*,

$$n_k^{\omega} = \sum_i \mathbb{I}[c_k^{\omega} = c_*^{\omega}(i)] \tag{4.12}$$

.

We then take the index k^* of the cluster with the most number of weights nearest to a cluster:

$$k^* = \text{argmax}_k \, n_k^{\omega} \tag{4.13}$$

Thus,

$$c_{k^*}^{\omega} \in \boldsymbol{c}^{\omega} = (c_1^{\omega}, \ldots, c_k^{\omega}) \tag{4.14}$$

is the cluster with the most number of weights nearest to it.

We then order the weights in W_{free}^t by their distance to $c_{k^*}^{\omega}$. In detail, for $W_{\text{free}}^t = [w_1, \ldots, w_i, \ldots, w_n]$, we reorder the weights by permuting the indices

$$w_i' = w_{\pi(i)} \tag{4.15}$$

where $\pi : [1, \ldots, n] \to [1, \ldots, n]$ is a permutation, $i \mapsto \pi(i)$. The ordered list $[w_1', \ldots, w_n']$ satisfies

$$D_{\text{prob}}(w_i', c_{k^*}^{\omega}) \leq D_{\text{prob}}(w_{i+1}', c_{k^*}^{\omega}) \tag{4.16}$$

Next, we need to determine how many of these weights we should assign to cluster $c^{\omega}_{k^*}$. To do so, we define a threshold δ and we take the first $\ell(\delta)$ weights from $[w'_1, ..., w'_n]$ such that:

$$\frac{1}{\ell(\delta)} \sum_{i=1}^{\ell(\delta)} D_{\text{prob}}(w'_i, c^{\omega}_{k^*}) \leq \delta. \tag{4.17}$$

As long as this is possible with $\ell(\delta) > 0$, we have identified both a cluster $c^{\omega}_{k^*}$ and set of weights $[w'_1, ..., w'_{\ell(\delta)}]$ which can be moved from W^t_{free} to W^{t+1}_{fixed}. We map the weights in $[w'_1, \ldots, w'_{\ell(\delta)}] = [(\mu'_1, \sigma'_1), \ldots, (\mu'_{\ell(\delta)}, \sigma'_{\ell(\delta)})]$ to a single weight

$$w_{k^*} = (\mu_{k^*}, \sigma_{k^*}) \tag{4.18}$$

corresponding to cluster $c^{\omega}_{k^*}$: $\mu_{k^*} = c^{\omega}_{k^*}$ and

$$\sigma_{k^*} = \texttt{std}([\mu'_1, \ldots, \mu'_{\ell(\delta)}]) \tag{4.19}$$

where `std` computes the standard deviation of its argument. This process is then repeated, finding the next most popular cluster until Np_t weights are assigned a cluster. If $\ell(\delta) = 0$, before enough weights are assigned in iteration t, then we have not been able to find any cluster centers $c_j \in \boldsymbol{c}^{\omega}$ which are close enough to any weight, *i.e.*, $D_{\text{prob}}(w_i, c_j) > \delta$ for all weights $w_i \in W^t_{\text{free}}$ and $c_j = c_{k^*}$. In this case, we set $\omega \leftarrow \omega + 1$ and $\delta \leftarrow 2\delta$ in the same step, giving us a higher-order additive powers-of-two set and less restrictive δ value threshold. Since $|\boldsymbol{c}^{\omega+1}| > |\boldsymbol{c}^{\omega}|$, this increase in ω makes more cluster centers available during the next clustering attempt.

4.3.6 Putting it All Together.

Putting the training and clustering stages together, we have a process for training a neural network whose weights are from a Gaussian posterior distribution with diagonal covariance matrix by backpropagation (BPP) that favours configurations with long Gaussian tails, which the clustering stage can then use to identify which weights to move and to what extent. This process is repeated for T iterations, with the cumulative fraction p_t of weights increasing with each t $p_{t+1} > p_t$ until all of the weights are moved from W_{free} to W_{fixed} at iteration T where $p_T = 1$.

After T iterations of training and clustering, each of the weights' distributions in the networks will have their μ values centered on one of the k clusters in the codebook.

Post the T fixing iterations there are two options depending on the downstream usage of the network: either the network can be converted into point estimates and the weights fixed to be the exact μ values giving us a quantised network. Or, we can use the extra information given to us by modelling each weight as a distribution as a way of quantifying uncertainty of a particular prediction. If, after multiple samples of $\boldsymbol{w}$, a model changes its prediction for a fixed input, this tells us that there is uncertainty in these predictions – with this information being useful for practical settings.

4.3.7 WFN to PWFN

The reader will have noticed some overlap between the formulation of WFN and PWFN, so let us step back to note the major differences.

4.3.7.1 The Proposal Set

In WFN, we found a recurrence relation $c_{j+1} = c_j(\frac{1+\delta}{1-\delta})$, $0 < \delta < 1$ that defined clusters which were guaranteed not to breach some distance threshold δ. We then took order ω additive-powers-of-two approximations of this set as our potential cluster set $\tilde{C}^{\omega}$. This was made possible since we had a fixed distance measure D_{rel} given a weights' magnitude.

This same formulation is not possible in PWFN since finding clusters centres equidistant from weights' would need to account for the variability in the distance measure due to the σ component. Instead, we just looked at *the entire* order ω additive-powers-of-two set as the pool of potential clusters C^{ω}. This will have the effect that $|\tilde{C}^{\omega}| \leq |C^{\omega}|$. The proposal cluster set in WFN will be the same size or smaller than the proposal set in PWFN since, in PWFN, we are not discounting clusters too far from the current weight set to be clustered.

4.3.8 On the Measure of Distance

Term Check

Euclidean Distance in the context of weights and clusters in a neural network, particularly when considering one-dimensional values, is simply the absolute difference between a weight w_i and a cluster center c_j. Mathematically, it is defined as:

$$D(w_i, c_j) = |w_i - c_j|$$

This is the simplest and most used distance measure in quantisation works.
Relative Distance (D_{rel}) is defined as the absolute value of the difference between two points normalized by one of the points. For a weight w_i and a cluster center c_j, it is:

$$D_{\text{rel}}(w_i, c_j) = \left| \frac{w_i - c_j}{w_i} \right|$$

This is the distance measure used in WFN.
Mahalanobis Distance (D_{prob}) in this context is redefined as the normalized difference between a mean μ_i and a cluster center c_j, scaled by the standard deviation σ_i. It is given by: $D_{\text{prob}}(w_i, c_j) = \left| \frac{\mu_i - c_j}{\sigma_i} \right|$ and used in the PWFN algorithm.

How we calculate distances from weights to clusters is the most salient change from WFN to PWFN. WFN measured the distance between a weight and a cluster as the weight magnitude normalised – the Manhattan distance. In contrast, in PWFN, we measure this as the Manhattan distances between the weight distribution mean and the cluster centre divided by the weight distribution standard deviation – Mahalanobis distance:

$$D_{\text{rel}} = \left| \frac{w_i - c_j}{w_i} \right| \qquad \text{vs.} \qquad D_{\text{prob}} = \left| \frac{\mu_i - c_j}{\sigma_i} \right| \tag{4.20}$$

The effect of this is that the training process can find configurations which decouple the weight magnitude and the distance that said weight could travel in the clustering stage.

4.3.9 To Prune or Not to Prune

One final area of variation from the original WFN lies with what to do about the weights with very extremely small magnitudes. Previously, we set a precision cap $b = -7$ for elements in C^s, and any value less than this was assigned to be closest to the zero clusters (using δ_0). In PWFN, we do not need to implement such a rule and can

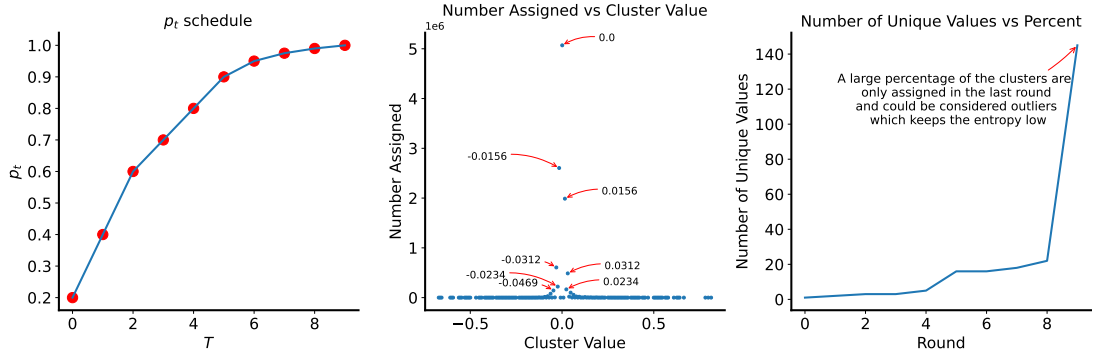


FIGURE 4.8: p_t follows the same schedule as [Subia-Waud and Dasmahapatra, 2022] (left). In the middle and right plots, we see that PWFN achieves very small entropy values by majority of weights to only a very small (4 or 5) cluster values and the rest are assigned as outliers, most of which are powers-of-two.

avoid having a fixed δ_0 if we want (although we do experiment with using one) and could allow weights $|\mu_i| < 2^{-7}$ to move away from zero depending on their σ_i value. The problem with doing so lies in the possibility of a situation where σ_i is extremely small, and no combination of elements in C^s up to a precision b could satisfy the δ threshold distance. In other words, no cluster centre could satisfy the distance threshold due to the precision restrictions.

The quandary has three solutions. We could allow an extremely large b-precision for the cluster centres, which, given a large enough order (number of additive powers-of-two components), could allow for clusters of ever greater precision, but this could enable far greater precision than is needed for solid results (as demonstrated in WFN). Instead, a simple solution is to increase the δ threshold multiplicatively by a hyper-parameter β as we increase the order, meaning that the distance threshold is relaxed for outliers. A final alternative is to maintain the functionality shown to work in WFN by using a δ_0 for which tiny values are clustered to zero. In our experiments, we will explore both of the latter cases.

4.3.9.1 The Highlighted Changes

In Algorithm 3, we redefine the clustering algorithm for a given iteration $t \in T$ with changes made from WFN highlighted in red.

4.4 Experiments

We conduct our experimentation on the ImageNet dataset with a wide range of models: ResNets-(18,34,50) [He et al., 2016], DenseNet-161 [Huang et al., 2017] and the

Algorithm 3: Clustering Np_t weights at the t^{th} iteration in PWFN.

while $|W^{t+1}_{\text{fixed}}| \leq Np_t$ **do**
  $\text{fixed}_{\text{new}} \leftarrow [\,]$
  while $\text{fixed}_{\text{new}}$ *is empty* **do**
    Increase the order $\omega \leftarrow \omega + 1$
    $c^{\omega} \leftarrow \{\sum_{i \in r} i \mid r \in \mathcal{P}(R) \wedge |r| \leq \omega\}$
    for each $i = 1 \ldots, |W^{t+1}_{\text{free}}|$
      $c^{\omega}_{*}(i) \leftarrow \min_{c \in C^{\omega}} D_{\text{prob}}(w_i, c)$
    for each cluster centre $c^{\omega}_{k} \in C^{\omega}$
      $n^{\omega}_{k} \leftarrow \sum_i \mathbb{I}[c^{\omega}_{k} = c^{\omega}_{*}(i)]$
    $k^{*} \leftarrow \arg\max_k n^{\omega}_{k}$
    Sort: $[w'_1, \ldots, w'_N] \leftarrow [w_1, \ldots, w_N]$, $w'_i = w_{\pi(i)}$, π permutation
      where $D_{\text{prob}}(w'_i, c^{\omega}_{k*}) < D_{\text{prob}}(w'_{i+1}, c^{\omega}_{k*})$
    $i \leftarrow 1$, mean $\leftarrow D_{\text{prob}}(w'_1, c^{\omega}_{k*})$
    while mean $\leq f(t, \delta_{t=0}, \delta_{t=T})$ **do**
      $\text{fixed}_{\text{new}} \leftarrow w'_i$
      mean $\leftarrow \frac{i}{i+1} \times$ mean $+ \frac{1}{i+1} \times D_{\text{prob}}(w'_{i+1}, c^{\omega}_{k*})$
      $i \leftarrow i + 1$
    $\delta_{t=t} \leftarrow \beta \times \delta_{t=t}$
  Assign all the weights in $\text{fixed}_{\text{new}}$ to cluster centre $c^{\omega}_{*}(i)$, moving them from W^{t+1}_{free} to W^{t+1}_{fixed}

challenging DeiT (small and tiny) [Touvron et al., 2021]. For each model, we convert all the parameters in the convolution and linear layers into Gaussian distributions where the mean value is set to be the weight value of the pre-trained model found in the Timm library. Thus, at test time with no further training, we retain the original accuracies. We set the variance parameters according to the setting described in Eq (4.7). We then apply nine rounds of the described weight fixing with three epochs of re-training each round, totalling to 27 epochs of training. We train using SGD with momentum 0.9 with a learning rate of 0.001. For all experiments, we fix $\delta = 1$, $\alpha = 2^{-11}$ which we found using grid-search on the CIFAR-10 dataset and works surprisingly well in all settings. For all our experiments we train using 4x RTX8000 GPUs and a batch-size of 128. For the ensemble results, we sample 20 times different weights using the learned weights' distributions and report the mean accuracy.

We further explore two settings:

1. PWFN (no prior): In the first setting, we do not use the prior initialisation and instead initialise the σ values with a random uniform distribution:

$$\sigma_i \sim \mathcal{U}(-0.0025, 0.0025) \tag{4.21}$$

TABLE 4.1: Full comparison results. (w/o FL-Bias) refers to calculating the metrics without the first-last layers and bias terms included. 'Params' refers to the unique parameter count in the quantised model, entropy is the full weight-space entropy. In-ch, layer, attn refer to whether the method uses a separate codebook for each layer, filter in-channel and attention head respectively.

		Separate Codebook					
Model	**Method**	**Layer**	**In-ch**	**Attn**	**Top-1** (Ensemble)	**Entropy**	**Params**
ResNet-18	Baseline	-	-	-	68.9	23.3	10756029
	LSQ		✗	-	68.2	-	-
	APoT		✗	-	69.9	5.7	1430
	WFN	✗	✗	-	69.7	3.0	164
	PWFN (no prior)	✗	✗	-	69.3 (69.6)	**1.7**	**143**
	PWFN	✗	✗	-	**70.0 (70.1)**	2.5	155
ResNet-34	Baseline	-	-	-	73.3	24.1	19014310
	LSQ		✗	-	71.9	-	-
	APoT		✗	-	73.4	6.8	16748
	WFN	✗	✗	-	73.0	3.8	233
	PWFN (no prior)	✗	✗	-	73.5 (74.4)	**1.2**	**147**
	PWFN	✗	✗	-	**74.3 (74.6)**	1.8	154
ResNet-50	Baseline	-	-	-	76.1	24.2	19915744
	LSQ		✗	-	75.8	-	-
	WFN	✗	✗	-	76.0	4.1	**261**
	PWFN (no prior)	✗	✗	-	77.2 (78.1)	3.5	334
	PWFN	✗	✗	-	**77.5 (78.3)**	**3.4**	325
DeiT-Small	Baseline	-	-	-	79.9	16.7	19174713
	LSQ+			✗	77.8	-	-
	Q-ViT				**78.1**	11.3	3066917
	Q-ViT (w/o FL-Bias)				**78.1**	10.4	257149
	PWFN (no prior)	✗	✗	✗	78.0 (78.3)	**2.7**	**352**
	PWFN	✗	✗	✗	**78.1 (78.5)**	**2.7**	356
DeiT-Tiny	Baseline	-	-	-	72.9	15.5	5481081
	LSQ+			✗	68.1	-	-
	Q-ViT				69.6	11.5	1117630
	Q-ViT (w/o FL-Bias)				69.6	10.5	128793
	PWFN (no prior)	✗	✗	✗	**71.4** (71.6)	**2.8**	300
	PWFN	✗	✗	✗	71.2 (71.5)	**2.8**	**296**
DenseNet161	Baseline	-	-	-	77.8	17.1	26423159
	PWFN	✗	✗	✗	77.6 (78.0)	1.1	125

2. PWFN: In the second setting, we use the informed prior outlined in Equation 4.7, setting the starting σ values as a weighted relative distance from their nearest power-of-two.

4.5 Results

We compare PWFN against a range of quantisation approaches where the model weights have been made available so that we can make accurate measurements of entropy and unique parameter count. For the ResNet family, we compare against the current state-of-the-art APoT [Yuhang Li, Xin Dong, 2020] [1] and WFN [Subia-Waud and

[1] https://github.com/yhhhli/APoT_quantisation

TABLE 4.2: Comparison of the number of additional training epochs required by different fine-tuning quantisation methods.

Method	Num of additional epochs
ApoT	120
PWFN	27
WFN	27
LSQ	90
QviT	300

Dasmahapatra, 2022] [2]. For the transformer models, there has only been one work released, Q-Vit [Li et al., 2022] [3], which has both the model saves and code released. For both APoT and Q-Vit, we compare the 3-bit models which are the closest in terms of weight-space entropy to that achieved by PWFN.

As presented in Table 4.2, PWFN requires substantially fewer additional training epochs than most methods, save for WFN, highlighting its training efficiency. We use a straightforward regularisation term that encourages an increase in σ, and its computational cost is comparable to that of L_1 regularisation. While our approach does lead to greater memory demands due to the additional σ parameters and their associated gradient updates, the overall simplicity of the method is more efficient than previous BNN training procedures, making it feasible to tackle more complex model-dataset pairings. Additionally, we note that when using the quantised version for inference, there are no extra costs. We take a sample (the μ values) from the BNN, effectively treating it as a point estimate.

Remember that WFN uses a regulariser that calculates the relative distance between all free weights and the existing cluster centers, and then penalises weights depending on the distance to their closest center (in a soft way). This incurs computational costs in the backpropagation calculation for every iteration. In the reformulated PWFN, we have a much simpler regularisation term that penalises sigma to increase - with costs that match that of l_0 regularisation. We do have memory overhead in terms of the number of parameters at training (σ and μ) and the random number generation to sample, but this is only at training time. The simplicity of the regularisation term also means that we experience a speed-up over the previous BNN training procedure outlined in the original Bayes-by-backprop paper, making much more complex model-dataset pairings tractable.

This is not to say there are no costs; we find that a single training epoch with ResNet-18 on the ImageNet dataset takes 1 hour 30mins on 4 consumer GPUs (GTX1080's) for PWFN, compared with 40 minutes for standard training and 1 hour 20mins for WFN (but for WFN this increases as the number of clusters increases through training).

[2] https://github.com/subiawaud/Weight_Fix_Networks
[3] https://github.com/YanjingLi0202/Q-ViT

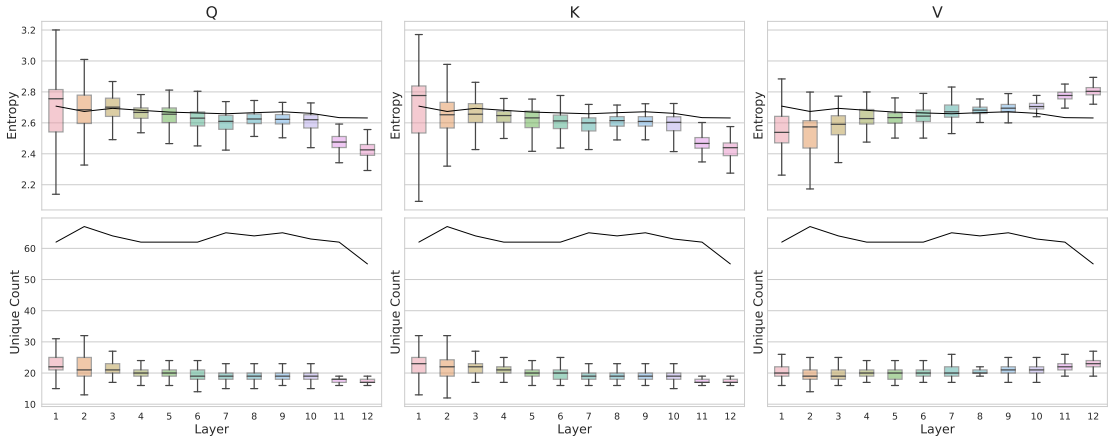


FIGURE 4.9: For DeiT small, we show a box plot of the entropies and unique counts per input channel for each Q,K,V by layer and with the mean of each layer (calculated across all attention heads) shown in the black lines.

In Table 4.1 we can see the set of results. PWFN demonstrates superior entropy, unique parameter count and top-1 accuracy across the board. In addition to the point-estimate accuracy using the mean of each of the weights' distributions (the cluster centers), we can additionally sample the weights from the learned distributions to give us an ensemble of models the mean prediction of which gives us further accuracy gains which we show in brackets in the Table. The prior initalisation gives a slight but consistent accuracy improvement over using a uniform prior (PWFN (no prior)). We note that for both APoT and Q-Vit different codebooks are used for different layers. For Q-Vit, different codebooks were additionally used for every attention head and input channel, and the bias terms were left unquantised, pushing up the parameter count and weight-space entropy substantially. As we have discussed in previous chapters, this is a growing trend in the field, where relaxations such as leaving large parts of the network unquantised, or using different codebooks for ever granular parts of the network, are often used. Each relaxation comes at a cost in hardware, be that support for unquantised elements – such as the first and last layers – or the use of different codebooks for various parts of the architecture.

4.5.1 Entropy Values by Layer

Figure 4.9 illustrates the variation in entropy and the count of unique parameters across different layers and attention components. A notable observation from our study is that the weights associated with the 'value' component exhibit higher entropy in the final layer. This observation aligns with the notion that employing a fixed quantisation scheme for each layer necessitates a relaxation of the quantisation constraints specifically for the last layer, as supported by prior studies [Yuhang Li, Xin Dong, 2020, Jung et al., 2019b, Zhou et al., 2016b, Yamamoto, 2021, Oh et al., 2021b, Li et al., 2022].

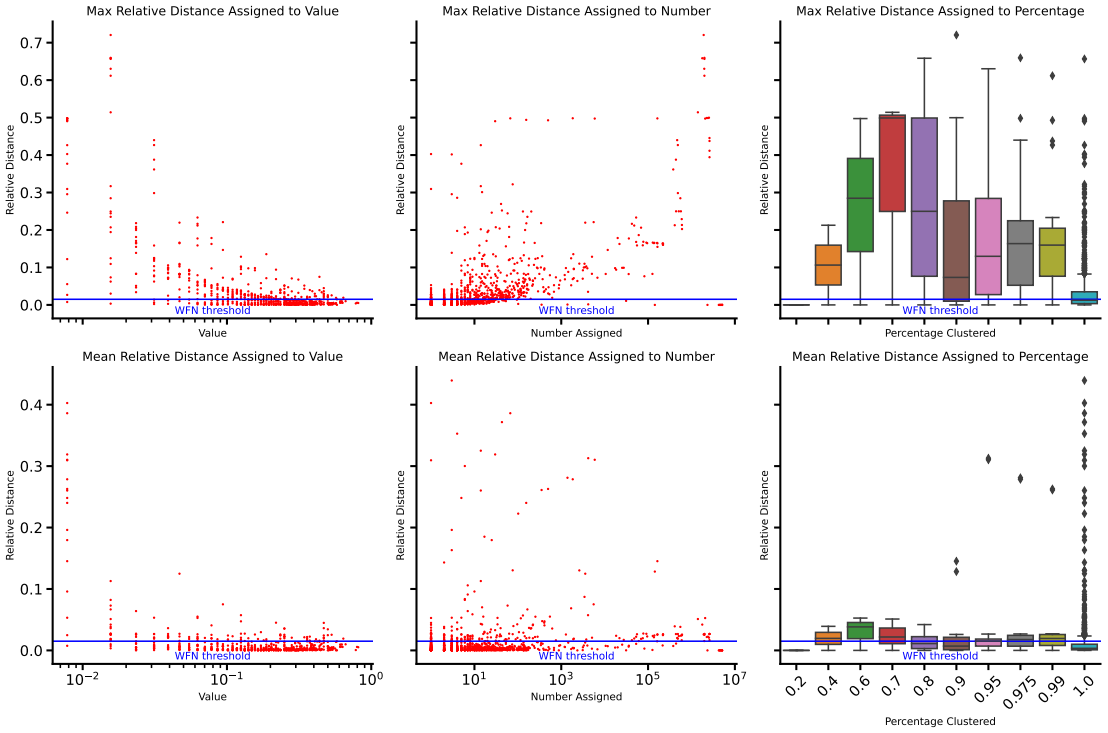


FIGURE 4.10: The maximum (top left) and mean (bottom left) relative distance a weight moves to a cluster by cluster value. The maximum relative distance is not well maintained with the number of weights assigned to that cluster (top middle), but the mean relative distance is (bottom middle). The maximum (top) relative distance for each cluster assignment and mean (bottom) relative distance by round are shown in the right-hand column. In all plots, we show in blue the threshold used in WFN.

Moreover, this highlights an intriguing possibility that in the context of attention networks, such relaxation might be essential only for the 'value' weights, and not for the 'keys' and 'queries'.

4.5.2 Distance Measures Compared

In understanding how PWFN is able to compress a network's representation to such a degree compared to WFN we look to how often the previously proposed relative distance threshold is maintained.

In Figure 4.10, it's evident that while the relative distance threshold established in WFN is, on average, maintained, there are edge-cases where it isn't. This observation suggests that having a context-specific noise tolerance benefits subsequent compression stages. Furthermore, the data indicates that these values are typically small (as seen in the left column), have a high frequency of occurrence (depicted in the middle), and are predominantly assigned during the middle (0.6, 0.7) and final rounds.
Let us now conclude on our findings of this Chapter.

4.6 Conclusion

PWFN offers something that WFN did not: position-specific signals for the clustering stage. This probabilistic reformulation and training of BNNs with weights coming from Gaussian distributions enabled us to use the weight uncertainty (σ) values in determining where to move specific weights. In doing so, we have seen that we have been able to further reduce the weight-space entropy and unique parameter count of a range of networks – including those with transformer networks containing attention layers – to a greater extent than current works. We have also understood why this is the case; upon examining the relative distance threshold used in WFN and comparing it with the distances acceptable for clustering/quantisation in PWFN, we observed a slight, but noticeable discrepancy. Some weights in WFN could be moved further than the relative distance allowed, which is only possible with position-specific weight distance thresholds made available through the probabilistic reformulation.

In addition, we have demonstrated for the first time that BNNs can be trained for more complex dataset-model pairings with a simple prior on the weight distributions, which maximises the values and thus the noise resilience of the network. This has allowed variational networks trained with backpropagation to match the performance of the point-estimate models for the ImageNet dataset for the first time.

Finally, we have achieved something of a corollary to our main objective of learning compressed network representations: we trained BNNs which have the inherent ability to provide uncertainty estimates and potentially be better calibrated. We will explore in a later chapter to what extent these uncertainty estimates provide utility over current approaches in this space. But for now, we turn to another area of potential improvement.

The current WFN and PWFN algorithms require separate training and clustering stages, where the clustering is not directly informed by gradient descent. In the next chapter, we look to reformulate the algorithm such that the gradient provides information on when best to cluster/quantise a weight, negating the need for separation of the two stages.

Chapter 5

Towards On-The-Fly Clustering in Weight Fixing Networks

In the preceding chapters, we have explored two methods for efficient weight quantisation in neural networks: Probabilistic Weight Fixing Networks (PWFN) and Weight Fixing Networks (WFN). Both methods focus on network compression by reducing the number of unique parameters in models, leading to more energy-efficient inference. However, their multi-stage implementation, which involves alternating between weight clustering and training to counteract errors introduced by earlier clustering stages using a fixed schedule, can be complex and may limit adaptability.

Currently, the transition from the training stage to the clustering stage is determined empirically after a predefined number of epochs. While this approach is practical, it lacks a strong theoretical foundation and may not always identify the optimal moment for clustering across various contexts. Furthermore, the choice regarding the number of clustering stages and the proportion of weights to be clustered is left to the discretion of the user, which can lead to variability in results.

In this chapter, we propose a more dynamic weight clustering method within the framework of Probabilistic Weight Fixing Networks. We introduce 'Cluster-On-the-Fly Probabilistic Weight Fixing Networks' (COF-PWFN), an approach that eliminates the need for predefined clustering stages. Instead, the optimisation process itself determines when and how to perform clustering. By analysing gradient data and distances to the nearest clusters, we aim to identify the appropriate time for a weight to undergo clustering.

The core idea of COF-PWFN is to maintain the Bayesian learning approach of PWFN while seamlessly integrating the clustering process into the training phase. This more integrated approach aims to simplify the overall process and potentially offer a more

consistent adjustment of weight values. By allowing the optimisation process to guide the clustering decisions, COF-PWFN seeks to adapt to the specific characteristics of each model and dataset, potentially motivated by more efficient and effective compression.

By introducing on-the-fly clustering, COF-PWFN aims to provide a more adaptive and streamlined approach to weight quantisation in neural networks. This chapter represents a step towards more dynamic and self-guided compression methods that can automatically adapt to the specific characteristics of each model and dataset, potentially leading to more efficient and effective deployment of deep learning models in resource-constrained environments.

5.1 On-The-Fly Clustering Approach (COF-PWFN)

Let us now outline the steps of COF-PWFN before providing a more formal treatment.

We initiate the COF-PWFN process by transforming a pre-trained model into its Bayesian Neural Network (BNN) counterpart, mirroring the PWFN approach. The model is then subjected to further training via gradient descent, where the loss function combines both cross-entropy loss and the large-σ promoting regularisation term. The key difference between COF-PWFN and PWFN lies in the determination of when to cluster weights and what percentage of weights to cluster. Both PWFN and WFN employ fixed epochs for training prior to clustering a predetermined percentage of the weights. Notably, these hyper-parameters are static and operate independently from the ongoing optimisation process.

In contrast, COF-PWFN continuously evaluates the clustering readiness of each weight after every training iteration based on two conditions. The first condition assesses a weight's contribution to the loss. If the gradient of the weight is sufficiently small, it indicates that the weight is optimally positioned on its local loss surface and could be clustered to a value within its positional vicinity. The second condition measures the weight's proximity to its designated cluster. For this, we utilise the sigma distance to the nearest cluster, an approach that proved effective in PWFN.

When both conditions are met for a weight, its value is updated to match the nearest cluster centre. This iterative cycle of gradient descent, weight evaluation, and clustering continues until all weights are clustered.

If the process stalls due to no weights meeting either clustering criterion after a training iteration, two potential interventions are considered: introducing additional cluster centres to facilitate further adjustments, or revising the clustering thresholds. These

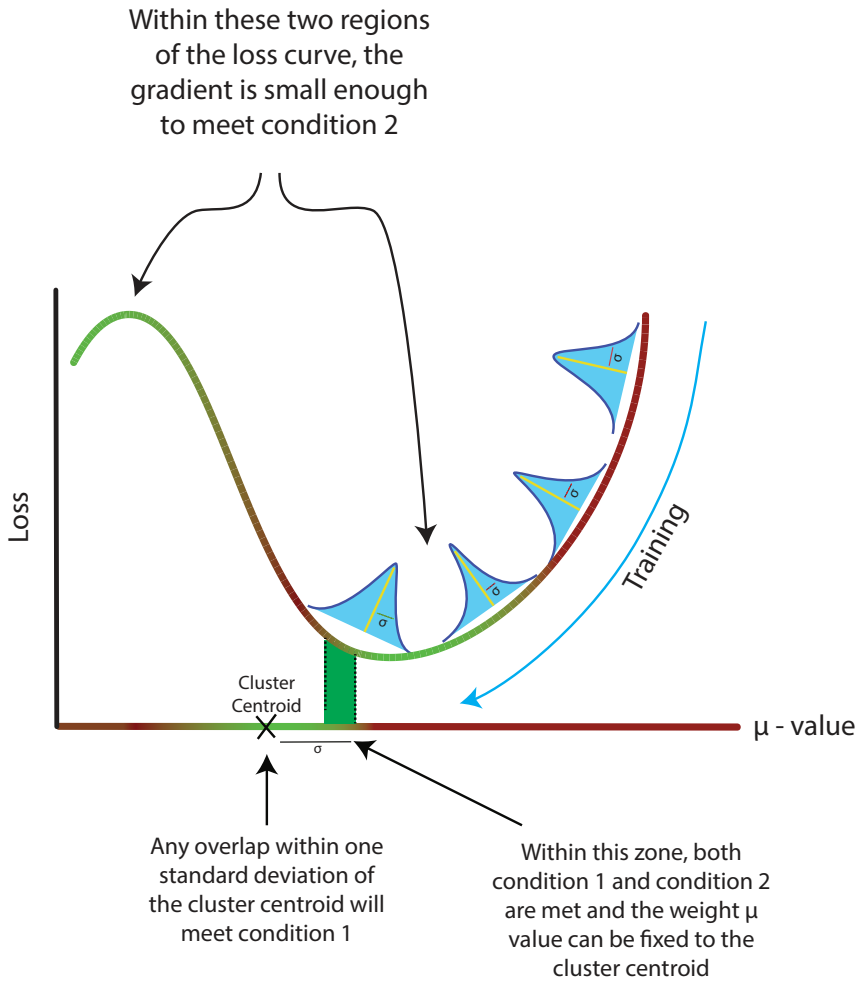


FIGURE 5.1: The two conditions needed to be met in order for a weight to be moved to a cluster are captured in this schematic. Condition 1 is met when the distribution of the weight has a small enough Mahalanobis distance to a cluster center. The cluster center is represented by an X in this diagram with the weight distribution moving through training represented with the blue normal distributions. Condition 2 is met when the weight-gradient is smaller than some threshold which we show in the diagram as the green sections of the loss curve.

adjustments have the overarching goal of enhancing the flexibility of the weight clustering mechanism.

Let us now turn to parsing out the details of the algorithm in more formal terms.

5.2 Method

Consider a neural network parameterised by N weights $\boldsymbol{w} = w_1, \ldots, w_N$. Within the COF-PWFN framework, each weight w_i is represented not by a single value, but is drawn from a distribution $w_i \sim \mathcal{N}(\mu_i, \sigma_i)$. The goal is to optimise the parameters $\boldsymbol{\mu} = (\mu_1, \ldots, \mu_N)$ and $\boldsymbol{\sigma} = (\sigma_1, \ldots, \sigma_N)$ of these distributions. During a forward pass in training, weight values w_i are sampled from their respective distributions as:

$$w_i = \mu_i + \sigma_i \epsilon, \epsilon \sim \mathcal{N}(0, 1). \tag{5.1}$$

Given fixed $\boldsymbol{\mu}$ and $\boldsymbol{\sigma}$, the forward pass remains stochastic due to the sampling of weight values from their respective distributions during each forward pass. Properly trained σ_i values offer insights into the noise resilience of the corresponding weight w_i. In essence, if an optimal configuration $\boldsymbol{w} = (\boldsymbol{\mu}, \boldsymbol{\sigma})$ can be found that upholds task performance amidst the variability introduced by σ_i, it provides knowledge about the flexibility of associated weights. The COF-PWFN methodology employs the Bayes-by-Backprop (BBP) optimisation technique, as described in [Blundell et al., 2015], with modifications in the initialisation and priors on $\boldsymbol{\mu}$ and $\boldsymbol{\sigma}$. The BBP technique is used to determine the parameters of the weight distributions by minimising a loss function that combines the negative log-likelihood of the data and a regularisation term that encourages larger $\boldsymbol{\sigma}$ values. The loss function used for training in COF-PWFN is given by:

$$-\log P(\mathcal{D}|\boldsymbol{\mu}, \boldsymbol{\sigma}) + \alpha \mathcal{L}_{\text{REG}}(\boldsymbol{\sigma}), \tag{5.2}$$

where the first term represents the negative log-likelihood of the data given the weight distribution parameters, and the second term is a regularisation term that encourages larger $\boldsymbol{\sigma}$ values. The regularisation term is defined as:

$$\mathcal{L}_{\text{REG}}(\boldsymbol{\sigma}) = \sum_{i=1}^{N} \mathcal{L}(\sigma_i) = -\sum_{i=1}^{N} (\sigma_i - S)\Theta(S - \sigma_i), \tag{5.3}$$

where $\Theta(x)$ is a step function that equals 1 for $x \geq 0$ and 0 otherwise, S is a constant that acts as a threshold for the σ_i values, and α is a hyperparameter that controls the strength of the regularisation. The role of the Θ function is pivotal; it deters the optimisation from producing a network with a subset of $\boldsymbol{\sigma}$ taking on disproportionately large values, which would overshadow the cross-entropy term. The constant S ensures $\boldsymbol{\sigma}$ stays within a reasonable range, while α guides the extent of pull towards larger σ's. Throughout the training iterations, each μ parameter is assigned a partial derivative, denoted $\frac{\partial L}{\partial \mu_i}$. This value, when scaled by an optimisation technique like Stochastic Gradient Descent or ADAM and influenced by a learning rate, dictates the direction and magnitude of change to reduce the loss. COF-PWFN uses this derivative to assess when

a weight is primed for clustering. Specifically, when $\frac{\partial L}{\partial \mu_i} < \sigma_{\text{grad}}$, with σ_{grad} being a pre-set gradient threshold, we postulate that the weight's position has minimal impact on the loss. The choice of σ_{grad} is based on the assumption that weights with small gradients are close to their optimal values and can be clustered without significantly affecting the model's performance. Such a weight is then deemed suitable to be "fixed" to a nearby position.

5.2.1 Defining *Close Proximity*

Consistent with the PWFN approach, we employ the strategy of depicting each weight as a Gaussian distribution and sampling during the forward pass based on its μ and σ parameters. This allows the σ values to serve as an indicator of the extent to which a weight might diverge from its established μ value.

With reference to the weight-space distance outlined in Equation 5.4, we introduce a threshold, σ_{dist}, for this distance. A weight is deemed proximate to a cluster centre c_k (defined subsequently) if the inequality $D_{\text{prob}}(w_i, c_j) < \sigma_{\text{dist}}$ holds true.

$$D_{\text{prob}}(w_i, c_j) = \frac{|\mu_i - c_j|}{\sigma_i}. \tag{5.4}$$

The metric $D_{\text{prob}}(w_i, c_j)$ – introduced in PWFN – quantifies the distance between a weight's mean μ_i and a cluster centre c_j in terms of the weight's standard deviation σ_i. This probabilistic distance measure takes into account the uncertainty associated with each weight, as captured by its σ value.

5.2.2 Clustering Conditions

Building upon the concept of close proximity, we introduce two conditions that must be satisfied for a weight to be clustered. These conditions pertain to each μ_i and σ_i in the network: ***condition one*** **:** $D_{\text{prob}}(w_i, c_j) < \sigma_{\text{dist}}$, and ***condition two*** **:** $\frac{\partial L}{\partial \mu_i} < \sigma_{\text{grad}}$. The first condition, *condition one*, ensures that the weight is sufficiently close to a cluster centre, as measured by the probabilistic distance metric $D_{\text{prob}}(w_i, c_j)$. This condition takes into account the uncertainty associated with the weight, as captured by its σ_i value, and compares it to the distance threshold σ_{dist}.

The second condition, *condition two*, assesses the impact of the weight on the loss function. By comparing the partial derivative $\frac{\partial L}{\partial \mu_i}$ to the gradient threshold σ_{grad}, we can determine whether the weight has converged to a stable position and is ready for clustering.

If a cluster c_j in the cluster set $\boldsymbol{c}$ satisfies both conditions, the weight μ_i is fixed to c_j. This weight is fixed for all future iterations with the ultimate aim to converge all μ_i's in the network to a compact cluster set $\boldsymbol{c}$.

5.2.3 Selecting Clusters

Having established the conditions for clustering weights, we now focus on determining the cluster set $\boldsymbol{c}$ within the COF-PWFN framework. The goal is to create a compact set of clusters that not only represents the distribution of $\boldsymbol{\mu}$ accurately but also promotes efficient multiplication by favouring powers-of-two values.

To ensure minimal size, it's imperative that the cluster set spans and represents the distribution of $\boldsymbol{\mu}$ effectively. This facilitates shorter distances to each μ_i, a crucial aspect for fulfilling the first clustering condition. We adopt a progressive strategy to introduce cluster values c_i into the weight cluster set $\boldsymbol{c}$. We initiate with a solitary value, $\boldsymbol{c} = \{0\}$, considering that the weight distribution in neural networks typically gravitates around zero. As training iterations ensue, we attempt to fix as many μ values as possible that meet the clustering conditions. If, during any iteration, no μ values meet the first condition while still satisfying the second, the cluster set $\boldsymbol{c}$ requires augmentation. Let's define R as the set of all powers-of-two up to a specified precision b:

$$R = -\frac{1}{2^b}, \ldots, -\frac{1}{2^{j+1}}, -\frac{1}{2^j}, 0, \frac{1}{2^j}, \frac{1}{2^{j+1}}, \ldots, \frac{1}{2^b} \,. \tag{5.5}$$

Consequently, the proposal cluster set $\boldsymbol{c}^\omega$ can be delineated as:

$$\boldsymbol{c}^\omega = \{\sum_{i \in r} i \mid r \in \mathcal{P}(R) \wedge |r| \leq \omega\} \setminus \boldsymbol{c}. \tag{5.6}$$

COF-PWFN starts with an order $\omega = 1$, considering the powers-of-two up to precision b as the initial proposal cluster set $\boldsymbol{c}^\omega$, excluding values already in $\boldsymbol{c}$. For each weight $w_i = (\mu_i, \sigma_i)$, we evaluate its distance from each cluster centre using the metric in Equation 5.4.

Given each free weight, we ascertain c_i^ω as the nearest cluster centre by distance measure. We then compute n_k^ω, the number of weights closest to cluster c_k^ω. The cluster with the most weights nearest to it, denoted as k, is incorporated into the cluster set $\boldsymbol{c}$. If the proposal set $\boldsymbol{c}^\omega$ is exhausted, we increase the order ω by one. The resulting mechanism prioritises the inclusion of all ω-additive powers-of-two before broadening the cluster set. This approach ensures that the cluster set $\boldsymbol{c}$ grows in a principled manner, adapting to the distribution of weights in the network while maintaining a focus on efficient multiplication (i.e lower bit-widths).

By integrating the concepts of close proximity, clustering conditions, and a principled cluster selection process, COF-PWFN provides a dynamic and adaptive framework for weight clustering in neural networks. The iterative nature of the algorithm allows for the gradual refinement of the cluster set $\boldsymbol{c}$, taking into account the evolving distribution of weights during the optimisation process. This approach aims to strike a balance between model compression and predictive performance, enabling the deployment of efficient neural networks in resource-constrained environments.

5.3 Algorithm

Let us finally put the constituent parts together with reference to Algorithms 4 and 5.

We initiate our BNN using pre-trained weights to set the mean values μ and, mirroring the PWFN setting, the standard deviations σ. We have gradient and distance thresholds defined as σ_{grad} and σ_{dist}, respectively. Additionally, we introduce a counter denoted as κ, initalised to zero, and a predefined parameter τ, which we refer to as the "patience" threshold. This additional term gives the algorithm some space for training to take place without needing to cluster at least one weight each training iteration up to τ.

After applying an iteration of training, for every weight parameter μ_i in the network, we evaluate its gradient with respect to the loss function and identify all weights not yet fixed that satisfy the condition that they are less than the predefined distance threshold σ_{dist} – as seen in lines 6-10 of Algorithm 4. Next, for each of these weights, we calculate their distances to each of the cluster centers in $\boldsymbol{c}$; if there exists a cluster center whereby a weight w_i has a computed distance of less than σ_{dist} (line 13) then we will fix the μ_i of said w_i (lines 14, 15).

However, if neither of the conditions ($\mathcal{C}_1$ and $\mathcal{C}_2$ refer to conditions one and two, respectively) is met for a weight, we increment the κ counter (lines 20-21). We continue the training iterations, and if the counter κ reaches or surpasses our patience threshold τ without any weights meeting the conditions, we take corrective actions: If the gradient condition is not met, we adjust the gradient threshold σ_{grad} (lines 27-28). If the distance condition is not met, we invoke the process detailed in Algorithm 5 (line 29-30). Once the corrective actions are taken, the κ counter is reset to zero.

The training concludes by updating the parameters μ and σ using an optimisation method, continuing this iterative process until all weights have been fixed to one of the values in the cluster set.

Algorithm 4: COF-PWFN training algorithm. The algorithm iteratively updates the weight distribution parameters μ and σ using gradient-based optimisation while simultaneously clustering weights based on their gradients and proximity to cluster centres. The gradient threshold σ_{grad} and the distance threshold σ_{dist} are adaptively adjusted to ensure effective clustering and convergence of the algorithm.

Input: Network parameters μ, σ, training data $\mathcal{D}$, gradient threshold σ_{grad}, distance threshold σ_{dist}, weight-space distance $D_{\text{prob}}(w_i, c_j)$, regularisation term $\mathcal{L}_{\text{REG}}(\sigma)$, learning rate α, threshold for allowable iterations τ

Output: Optimised parameters μ, σ and cluster set c

Initialize μ, σ, $c = \{0\}$, and $\kappa = 0$;
$w_{\text{free}} = \mu$;
while *not all parameters fixed* **do**
    Compute the loss function using Equation 5.2;
    Compute the partial derivatives $\frac{\partial L}{\partial \mu_i}$ for $\mu_i \in w_{\text{free}}$ and $\frac{\partial L}{\partial \sigma_i}$ for all σ;
    Update μ_i for $\mu_i \in w_{\text{free}}$ and σ_i for all σ using optimizer;
    $\mathcal{C}_1 = 1$; /* Condition 1 flag */
    $\mathcal{C}_2 = 1$; /* Condition 1 flag */
    foreach $\mu_i \in w_{free}$ **do**
        if $\frac{\partial L}{\partial \mu_i} < \sigma_{grad}$ **then**
            $\mathcal{C}_1 = 0$; /* Condition 1 satisfied */
            foreach c_j *in* c **do**
                if $D_{prob}(w_i, c_j) < \sigma_{dist}$ **then**
                    Fix weight μ_i to c_j;
                    Remove μ_i from w_{free};
                    $\mathcal{C}_2 = 0$; /* Condition 2 satisfied */
                end
            end
        end
    end
    if $\mathcal{C}_1 = 1$ ***or*** $\mathcal{C}_2 = 1$ **then**
        $\kappa = \kappa + 1$; /* Increase count */
    else
        $\kappa = 0$;
    end
    /* Is the count greater than the allowable iterations without any fixing */
    if $\kappa \geq \tau$ **then**
        if $\mathcal{C}_1 = 1$ **then**
            Multiply σ_{grad} by 2;
        else
            Call Algorithm 5: COF-PWFN Adding a Cluster;
        end
        $\kappa = 0$;
    end
end

Algorithm 5: COF-PWFN Adding a Cluster

Input: Free weights W^t_{free}, current cluster set c, maximum precision b, order ω
Output: Updated cluster set c, order *omega*

Let $R = \{-\frac{1}{2^b}, \ldots, -\frac{1}{2^{j-1}}, -\frac{1}{2^j}, 0, \frac{1}{2^j}, \frac{1}{2^{j+1}}, \ldots, \frac{1}{2^b}\}$;
Initialize $c' = \{\sum_{i \in r} i \mid r \in \mathcal{P}(R) \wedge |r| \leq \omega\} \setminus c$;
if $|c'| = 0$ **then**
    Increase order: $\omega = \omega + 1$;
    Update $c' = \{\sum_{i \in r} i \mid r \in \mathcal{P}(R) \wedge |r| \leq \omega\} \setminus c$;
end
foreach $w_i = (\mu_i, \sigma_i) \in W^t_{free}$ **do**
    Compute $D_{\text{prob}}(w_i, c)$ for $c \in c'$;
    Define $c^\omega_*(i) = \min_{c \in c'} D_{\text{prob}}(w_i, c)$;
end
Define $n^\omega_k = \sum_i \mathbb{I}[c^\omega_k = c^\omega_*(i)]$;
Find $k^* = \text{argmax}_k \, n^\omega_k$;
Update $c = c \cup \{c^\omega_{k^*}\}$;
Return c, ω;

5.4 Experiments and Results

In this section, we evaluate the performance of COF-PWFN on both the CIFAR-10 and ImageNet datasets using the ResNet family of models. We first explore the hyperparameter space for σ_{grad}, σ_{dist}, and τ using a grid search with a ResNet-18 model on the CIFAR-10 dataset. The results of this exploration are presented in Figure 5.2. Following this, we apply COF-PWFN to larger-scale experiments on the ImageNet dataset using various ResNet models and compare its performance with other state-of-the-art quantisation methods.

5.4.1 Hyperparameter Exploration

The grid search over the hyperparameters σ_{grad}, σ_{dist}, and τ reveals several notable trends, as shown in Figure 5.2. The most intuitive observation is that increasing σ_{grad} (left column of Figure 5.2) leads to a decrease in accuracy, entropy, and the epoch at which all parameters are fixed. This behaviour can be attributed to the fact that fixing weights when they still have large gradients can result in a smaller pool of clusters, but at the cost of declining network performance. Essentially, a higher σ_{grad} value allows weights to be clustered more aggressively, even if they have not fully converged, leading to suboptimal clustering and reduced accuracy. The impact of the patience threshold τ is somewhat less intuitive. An increase in τ can result in a decline in accuracy, entropy, and convergence epoch, but in a less predictable manner with higher variability. Interestingly, even with $\tau = 1$, a drop in accuracy is observed, suggesting that this

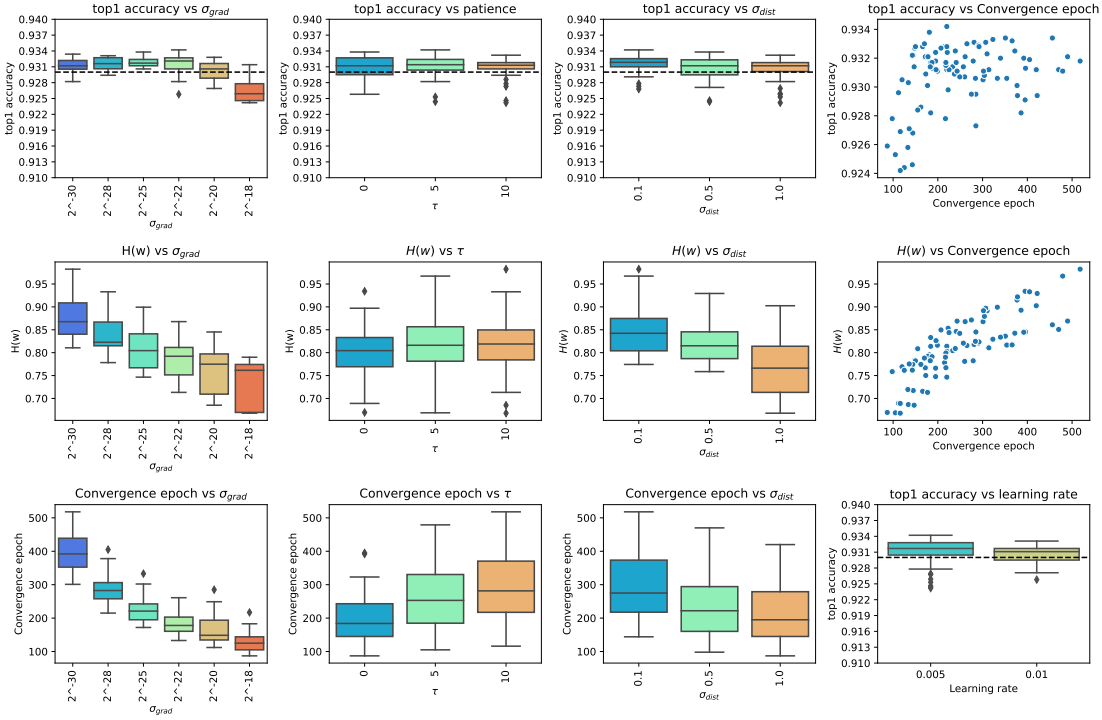


FIGURE 5.2: Hyperparameter exploration using ResNet-18 trained on the CIFAR-10 dataset.

additional layer of complexity may not be necessary. Consequently, we fix $\tau = 0$ for the larger-scale ImageNet experiments to simplify the algorithm and reduce computational overhead. With $\tau = 0$ at every training iteration at least one weight will be fixed.

5.4.2 ImageNet Results

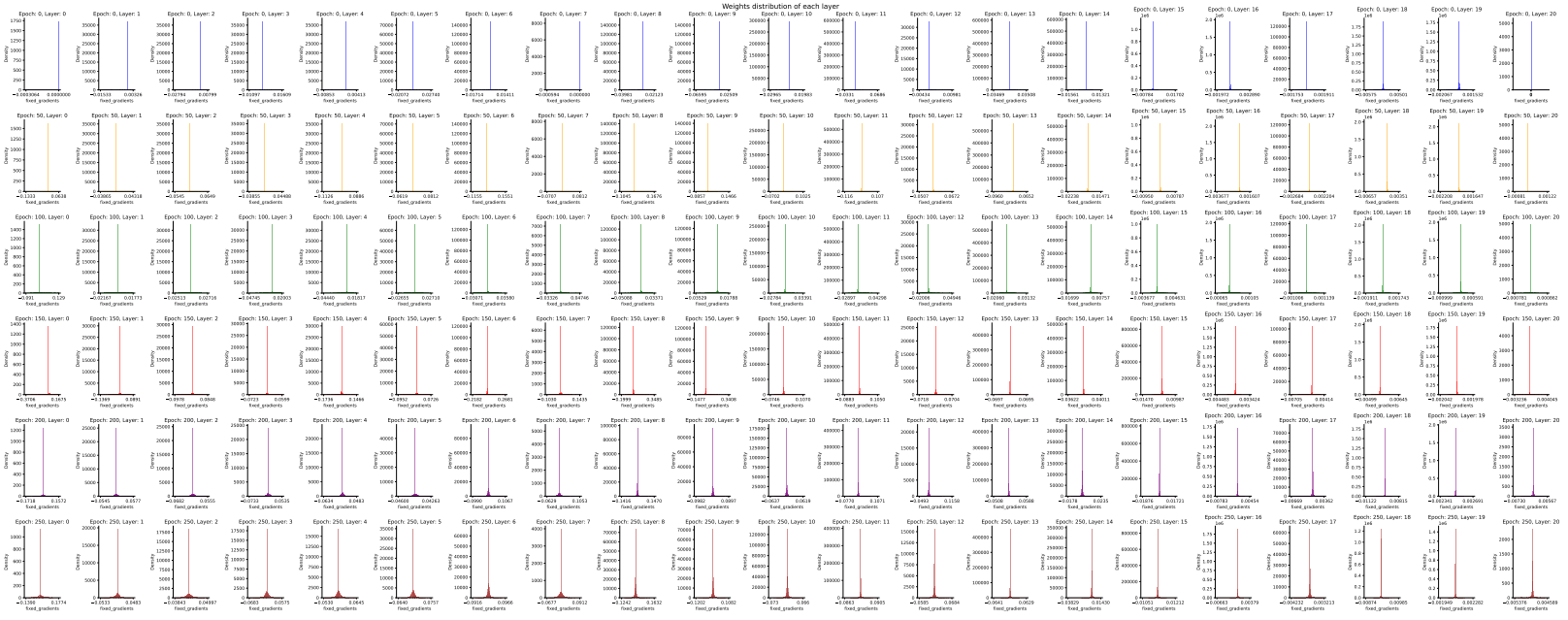
We present the full results comparing COF-PWFN with WFN, PWFN, LSQ, and APoT for the ImageNet dataset using fixed hyperparameters found to be optimal in the CIFAR-10 experiments in Table 5.1.

The results show that while COF-PWFN outperforms all other techniques in terms of accuracy, it does so with not as strong compression rates compared to both WFN and PWFN. This suggests that there may be a trade-off between accuracy and compression in the current hyperparameter settings. However, we hypothesise that there exists a hyperparameter configuration that can achieve both high accuracy and strong compression rates, potentially matching or even surpassing the performance of PWFN. To find such optimal hyperparameter settings, several strategies can be employed. One approach is to conduct a more extensive grid search over a wider range of hyperparameter values. This would allow us to explore a larger portion of the hyperparameter space and potentially identify configurations that strike a better balance between accuracy and compression. Another strategy is to use more advanced

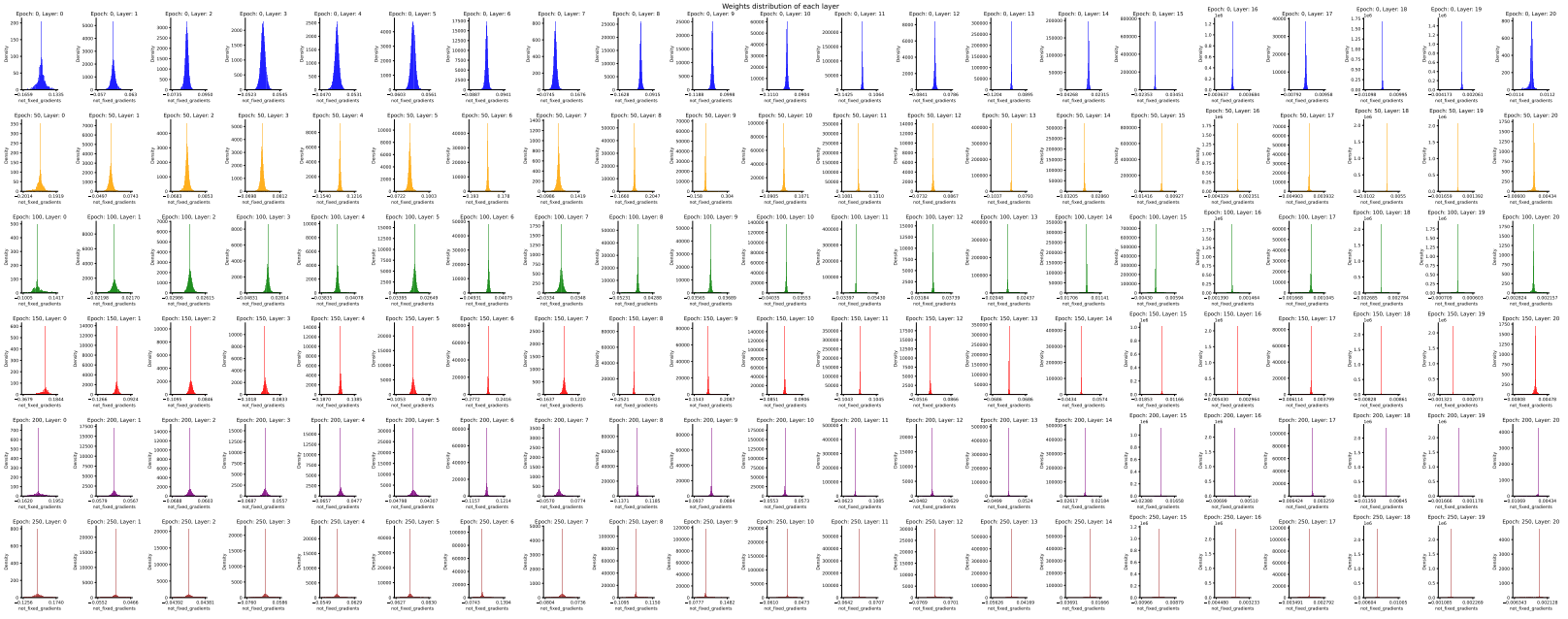
Model	Method	**Top-1** (Ensemble)	Entropy	Params	Converg
ResNet-18	Baseline	68.9	23.3	10756029	-
	LSQ	68.2	-	-	90
	APoT	69.9	5.7	1430	120
	WFN	69.7	3.0	164	27
	PWFN (no prior)	69.3 (69.6)	**1.7**	**143**	27
	PWFN	70.0 (70.1)	2.5	155	27
	COF-PWFN	**70.1 (71.8)**	4.1	292	33
ResNet-34	Baseline	73.3	24.1	19014310	-
	LSQ	71.9	-	-	90
	APoT	73.4	6.8	16748	120
	WFN	73.0	3.8	233	27
	PWFN (no prior)	73.5 (74.4)	**1.2**	**147**	27
	PWFN	74.3 (74.6)	1.8	154	27
	COF-PWFN	**74.4 (76.7)**	3.6	285	36

TABLE 5.1: A comparison set of quantisation results for the Imagenet dataset. Params refers to the number of unique parameters in the network and Converg is number of epochs used in the fine-tuning stage.

hyperparameter optimisation techniques, such as Bayesian optimisation or evolutionary algorithms, which can efficiently search the hyperparameter space by leveraging information from previous evaluations. Furthermore, analysing the relationship between the hyperparameters and the performance metrics can provide valuable insights into the behavior of COF-PWFN. By examining how accuracy and compression rates vary with different hyperparameter settings, we can gain a deeper understanding of the trade-offs involved and identify potential sweet spots in the hyperparameter space. This analysis can also guide the development of more efficient search strategies and help prioritise the most promising hyperparameter configurations.



(a) Fixed Gradients Distribution



(b) Not Fixed Gradients Distribution

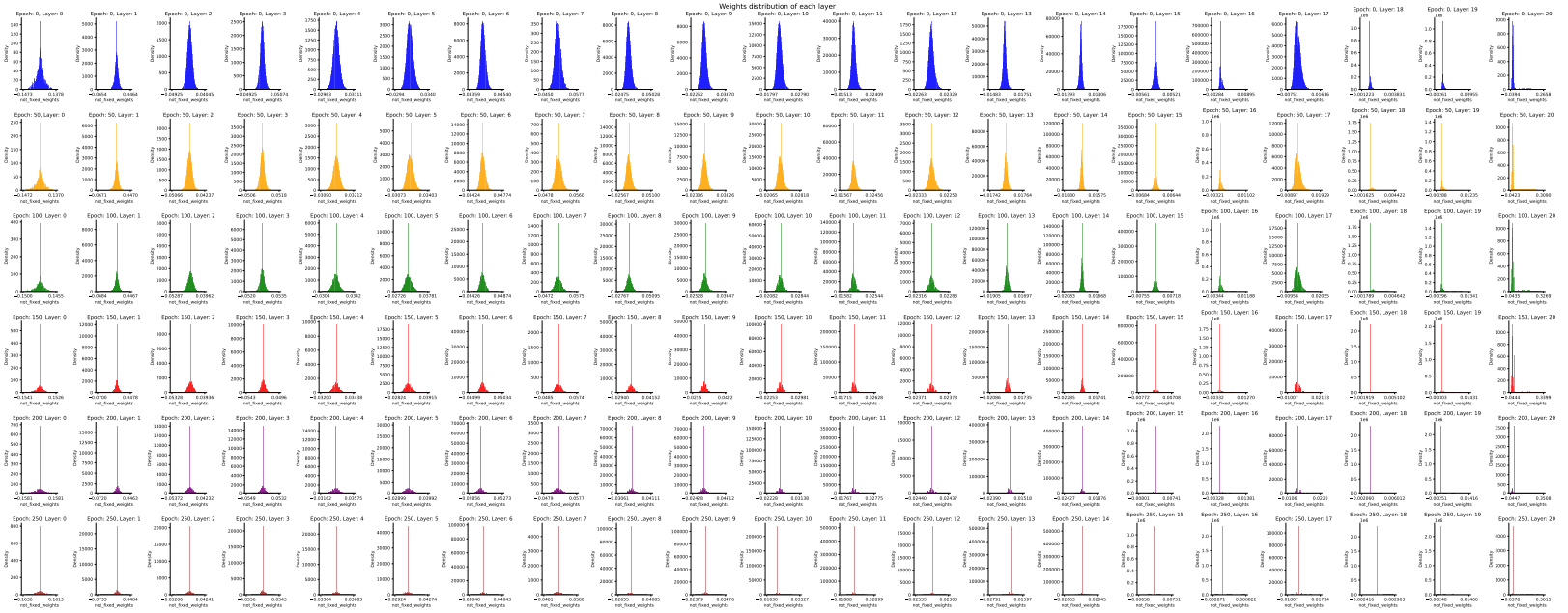
TABLE 5.2: Comparison of fixed and not fixed distributions of gradients as they change over time with each epoch for each of the different layers.

To gain further insights into the behaviour of COF-PWFN during training, we analyse the distributions of gradients and weights for both fixed and not-fixed parameters across different layers and epochs. Figure 5.2 presents a comparison of the fixed and not-fixed distributions of gradients, whilst Figure 5.3 illustrates the corresponding distributions for weights. These visualisations reveal several interesting patterns. Firstly, the gradients of fixed parameters build up, first with just a zero cluster and then adding powers-of-two before the outliers. We can see additionally that the last and first layers follow a very different pattern to clustering to the others, demonstrating why prior works have opted to leave these two layers unquantised. What works in quantising middle layers requires a different set-up for the first and last layers.

We can obtain gradient information for fixed weights because during inference, gradient calculations are performed for all weights, both fixed and not fixed. Subsequently, we mask the gradients of fixed weights to zero. Examining the gradient dynamics in Figure 5.2, we observe that once fixed, the gradient pressure on these weights remains small. This suggests that the clustered weights are in 'good positions' and do not require adjustment to reduce the loss. However, as we progress through epochs, some outliers emerge in the fixed weight gradients. These outliers indicate some pressure and might point to a potential improvement: allowing certain weights to depart from their cluster centres if sufficient gradient pressure accumulates.

While computational constraints limit our ability to run extensive experiments in this study (ResNet-50 was just out of reach due to memory limitations), the results presented here demonstrate is it indeed possible to compress to – prior to WFN and PWFN – SOTA levels without any drop in performance without the iterative rounds outlined in PWFN and WFN. Next, we turn to the question of the utility, if any, of the σ parameters learned in PWFN and COF-PWFN in determining the confidence and calibration of predictions with this family of models.

(a) Fixed Weights Distribution



(b) Not Fixed Weights Distribution

TABLE 5.3: Comparison of fixed and not fixed distributions of weights as they change over time with each epoch for each of the different layers.

Chapter 6

Uncertainty Estimations of BWFN

Thus far in this thesis, we have proposed three novel techniques for neural network compression, two of which utilised a stochastic reformulation of pre-trained networks into Bayesian Neural Networks (BNNs). The weight distributions of the BNNs were able to be used as signal for the downstream compression algorithm, and was shown to be a boon for increasing the accuracy through ensemble sampling. A yet unexplored avenue pertains to the potential of the PWFN and COF-PWFN reformulations for uncertainty estimations.

In this chapter, we turn to this question as we investigate the capability of the uncertainty distributions learned by these models to serve as reliable estimations of uncertainty in neural networks. Before we begin our jaunt into some experiments to test the utility of the learned BNNs, let's motivate our study and look at why uncertainty estimations are so useful and where current research directions have been focussed.

6.0.1 The Landscape of Uncertainty Estimations

The essence of uncertainty estimation in neural networks can be distilled into several critical considerations:

- Safety Critical: Neural networks are often used in tasks where decision-making is critical, such as autonomous driving or medical diagnosis [McAllister et al., 2017, Begoli et al., 2019, Muhammad et al., 2020, Jungo et al., 2018]. Uncertainty estimation provides valuable information about the confidence or reliability of the model's predictions [Gal and Ghahramani, 2016]. This information can be used to make more informed decisions or trigger appropriate actions when uncertainty is high [Guo et al., 2017]. Moreover, in scenarios where neural networks collaborate

with humans, expressing uncertainty can improve trust and transparency [Ribeiro et al., 2016].

- Distribution detection: Neural networks are typically trained on a specific distribution of data. However, during deployment, they may encounter inputs that differ significantly from the training data [Hendrycks and Gimpel, 2016]. Uncertainty estimation can help identify such out-of-distribution samples by detecting high uncertainty levels [Lakshminarayanan et al., 2017]. This is crucial for avoiding erroneous predictions or flagging inputs that fall outside the model's intended use case [Guo et al., 2017].

- Model prediction trust: The trustworthiness of a neural network relies heavily on its ability to provide accurate confidence estimates that align with its true accuracy [Guo et al., 2017]. Model calibration is crucial because it ensures that the predicted probabilities or confidence scores genuinely reflect the true likelihood of correctness [Niculescu-Mizil and Caruana, 2005]. In many real-world scenarios, especially where decisions have significant consequences, relying on miscalibrated predictions can lead to suboptimal or even harmful decisions [Platt et al., 1999]. Therefore, proper calibration is essential for ensuring reliable and interpretable results.

- Active learning and data acquisition: Uncertainty estimation is fundamental in guiding the active learning process, helping in the selection of informative or challenging samples for labeling [Settles, 2011]. By recognising instances with high uncertainty, neural networks can prioritize acquiring labels for those samples that are expected to provide the most significant improvement to model performance [Gal et al., 2017]. This approach ensures not only an efficient labeling process but also an optimized use of resources and an enhancement in overall model accuracy with potentially fewer labeled instances [Roy and McCallum, 2001].

6.0.1.1 Uncertainty Estimation in Deep Learning

The field of DNNs has witnessed significant advancements in uncertainty estimation. These developments span a range of methodologies, each addressing the complexities of uncertainty in unique ways.

Bayesian Framework: The Bayesian framework remains foundational in this realm. As we have seen, BNNs treat network weights as probability distributions, offering a comprehensive framework for uncertainty estimation [Blundell et al., 2015]. Complementing this, Bayesian Dropout, as articulated by Gal and Ghahramani, interprets dropout layers in a Bayesian context, providing an efficient means of uncertainty approximation [Gal and Ghahramani, 2016, Kendall and Gal, 2017] without the need to train full-Bayesian models. The reason this this

approach can be a potential boon for uncertainty estimation is the idea that different parameter settings (from sampling) gives us an output distribution on a particular input sample. This output distribution can tell us something about the uncertainty in our predictive ability for the given sample.

Ensemble Methods: Deep Ensembles, proposed by Lakshminarayanan et al., train multiple neural networks independently, capturing diverse outcomes and uncertainties effectively [Lakshminarayanan et al., 2017, Tagasovska and Lopez-Paz, 2019]. Stochastic Weight Averaging (SWA) and its Gaussian variant (SWAG) average model weights over training or approximate the weight distribution, enhancing the robustness of uncertainty estimates without needing to train separate networks from scratch for the ensemble [Izmailov et al., 2018, Maddox et al., 2019].

Post-Training Metrics: Recent studies have focused on post-training metrics for uncertainty estimation. Notable work in medical image segmentation examined calibration methods like Platt scaling and fine-tuning applied after training to recalibrate uncertainty estimates [Rousseau et al., 2021]. Training-free methods, such as infer-transformation and infer-noise, offer flexible, efficient approaches without retraining the model [Mi et al., 2019].

Reinforcement Learning and Uncertainty: The exploration of uncertainty in reinforcement learning has led to innovative methodologies like the bootstrapped DQN, offering deeper insights into the dynamics of uncertainty within these frameworks [Osband et al., 2016].

Adversarial Contexts: Work has looked at the interplay between adversarial contexts and uncertainty and found this key in understanding the robustness of deep learning models [Smith and Gal, 2018].

Distribution-Agnostic Methods: Efforts have been made towards devising distribution-agnostic techniques for uncertainty prediction, aiming to predict uncertainty without being tethered to a specific distribution [Pearce et al., 2018].

Despite these advancements, challenges persist, particularly in terms of computational efficiency and integration into existing workflows. For a full discussion of the merits, rationale and current state-of-the-art we point the reader [Gawlikowski et al., 2021].

Let's now shift our focus to defining popular metrics used to characterise how well calibrated models are from samples to predictive uncertainty. These metrics broadly aim to gauge the alignment of a method with the ground-truth uncertainty estimation outputs.

6.1 Evaluation Metrics

There has been a number of evaluation metrics that look to capture uncertainty estimation capabilities of models [Vaicenavicius et al., 2019]. Let us take a brief dive into a selection and their definitions:

6.1.0.1 Brier Score

The Brier Score [Brier, 1950] is a proper score function that measures the accuracy of probabilistic predictions. It is applicable to both binary and multi-class prediction problems.

To calculate the Brier score for a binary classification problem, given:

- y_i: The actual outcome. It is 1 if the outcome is positive and 0 if it's negative.
- $\hat{y}_i$: The predicted probability of the outcome being positive.

The Brier score (BS) for binary classification problems can be defined as:

$$BS = \frac{1}{N}\sum_{i=1}^{N}(y_i - \hat{y}_i)^2 \tag{6.1}$$

Where:

- N is the total number of predictions.
- The summation $\sum_{i=1}^{N}$ runs over all predictions.
- $(y_i - \hat{y}_i)^2$ is the squared difference between the actual outcome and the predicted probability for each prediction.

For multi-class classification problems, assuming C is the number of classes, and $\hat{y}_{i,c}$ and $y_{i,c}$ are the predicted probability and actual outcome (1 if it is true, 0 otherwise) of class c for observation i, the Brier score is:

$$BS = \frac{1}{N}\sum_{i=1}^{N}\sum_{c=1}^{C}(y_{i,c} - \hat{y}_{i,c})^2, \tag{6.2}$$

where:

- The inner summation $\sum_{c=1}^{C}$ calculates the squared differences for each class.
- The outer summation $\sum_{i=1}^{N}$ averages these values over all observations.

Interpretation of Brier Score:

The Brier Score essentially computes the mean squared difference between predicted probabilities and the actual outcomes. Its value ranges from 0 to 1, with the following implications:

- $BS = 0$: Represents perfect predictions, meaning all probabilistic forecasts match the actual outcomes perfectly.
- $BS = 1$: Represents the worst-case scenario, where all predictions are completely opposite to the actual outcomes.
- $0 < BS < 0.25$: Typically indicates high accuracy in the probabilistic forecasts.
- $0.25 \leq BS \leq 0.5$: Suggests moderate accuracy.
- $BS > 0.5$: Indicates low accuracy and potential issues with the model or the predictions.

In essence, the Brier Score quantifies the reliability of probabilistic predictions. A lower Brier Score indicates better predictive accuracy, while a higher score suggests potential calibration issues. It provides a comprehensive view of a model's performance, capturing both calibration and refinement in a single metric.

6.1.1 Reliability Diagrams

Reliability diagrams serve as invaluable tools for visually assessing the calibration of predictive models [Niculescu-Mizil and Caruana, 2005, Platt et al., 1999, Guo et al., 2017, Naeini et al., 2015]. The primary objective is to juxtapose the expected accuracy against the confidence of predictions.

Given:

- B_m: The set of indices corresponding to samples categorized into bin m.
- p_i: The predicted probability associated with sample i.
- y_i: The actual outcome for sample i, where a value of 1 denotes a positive outcome and 0 indicates a negative one.

For each designated bin m, the calculations proceed as:

- The average predicted probability, denoted as $\text{conf}(B_m) = \frac{1}{|B_m|} \sum_{i \in B_m} p_i$.
- The accuracy of predictions within the bin, represented as $\text{acc}(B_m) = \frac{1}{|B_m|} \sum_{i \in B_m} y_i$.

Subsequently, a plot is constructed with $\text{conf}(B_m)$ on the x-axis against $\text{acc}(B_m)$ on the y-axis, spanning all bins m.

Interpretation of Reliability Diagrams:

Reliability diagrams (as shown in Figure 6.1 provide a vivid representation of a model's calibration performance, facilitating the following interpretations:

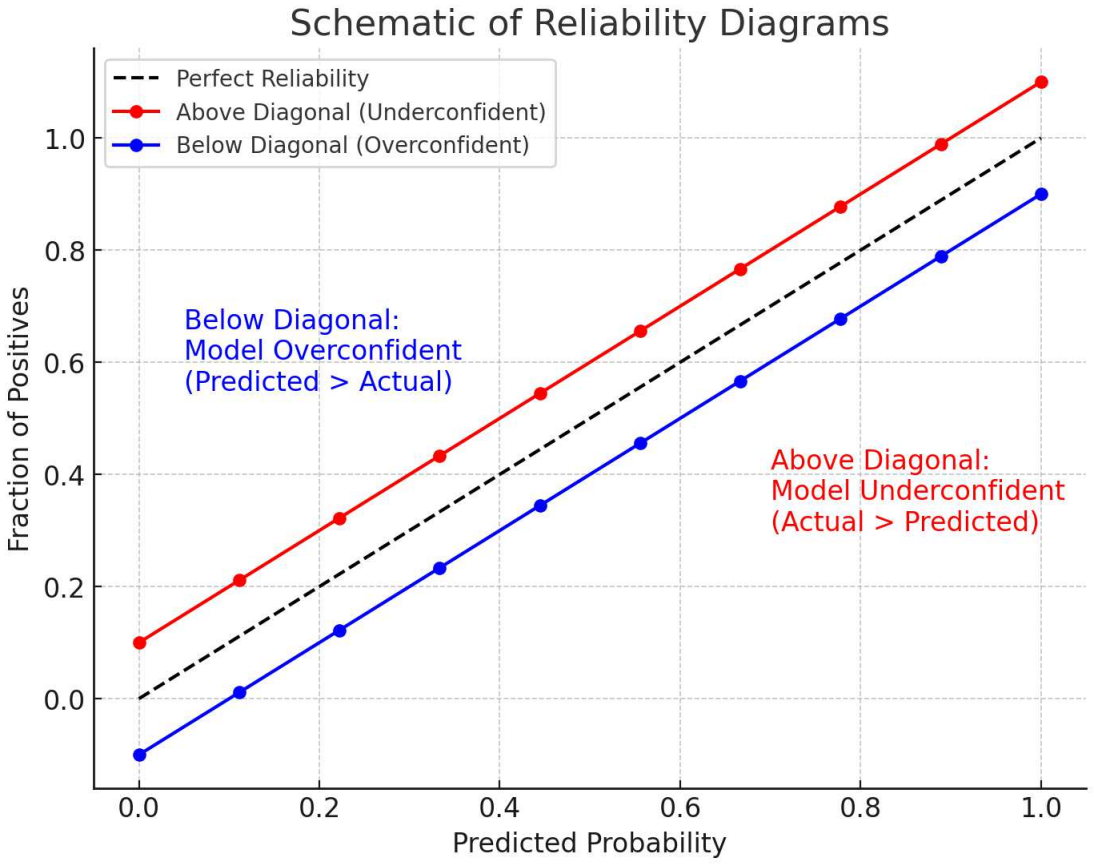


FIGURE 6.1: This schematic of a reliability diagram visually illustrates the calibration of probabilistic predictions in binary classification. The X-axis represents the predicted probabilities by the model, while the Y-axis shows the actual fraction of positive outcomes. The dashed line indicates perfect reliability, where predictions perfectly match observed outcomes. Points above the diagonal (red) signify underconfidence, indicating that the actual fraction of positives is higher than predicted. Conversely, points below the diagonal (blue) denote overconfidence, where predicted probabilities surpass the actual fraction of positives.

- A perfectly calibrated model is depicted by a diagonal line from the bottom-left to the top-right corner. This signifies that for every predicted probability, the fraction of positive outcomes matches the predicted probability.
- Deviations from the diagonal represent discrepancies between the predicted probabilities and the observed frequencies. Such deviations signal miscalibration. For instance, if for a predicted probability of 0.7, the observed frequency is 0.5, it suggests that the model's predictions are overconfident.
- Bins that lie above the diagonal indicate that the model is underconfident in its predictions for that bin. Conversely, bins below the diagonal suggest overconfidence.
- The granularity of the bins can also offer insights. For instance, if a model only makes predictions with extreme confidences close to 0 or 1, this could

be represented by sparse bins in the center of the diagram, indicating potential overfitting or a lack of nuance in the model's predictions.

– The width of the bins and the number of samples in each bin can affect the reliability diagram's appearance. Fewer, broader bins can mask miscalibrations that might be evident with more, narrower bins.

In summary, reliability diagrams serve as a potent tool for discerning the calibration quality of models. They offer a visual means to detect regions of overconfidence or underconfidence, facilitating informed adjustments to enhance the model's predictive quality.

6.1.2 Maximum Calibration Error (MCE)

The Maximum Calibration Error (MCE) [Niculescu-Mizil and Caruana, 2005] provides a scalar measure to quantify the calibration quality of probabilistic classification models. It specifically addresses the largest discrepancy between predicted probabilities and observed outcomes across all bins in a reliability diagram.

Given a set of bins $B_1, B_2, \ldots, B_M$ in a reliability diagram, where:

– $\text{conf}(B_m)$ is the average predicted probability for bin m.

– $\text{acc}(B_m)$ is the accuracy of bin m.

The MCE is formally defined as:

$$\text{MCE} = \max_{m=1,\ldots,M} |\text{conf}(B_m) - \text{acc}(B_m)| \tag{6.3}$$

Interpretation of MCE:

The MCE captures the worst-case error between the model's confidence and its actual performance in any of the bins. A lower MCE indicates better calibration, implying that the model's predicted probabilities align more closely with the actual outcomes. On the other hand, a higher MCE signifies that there exists at least one bin where the model's predicted probabilities significantly deviate from the actual outcomes, suggesting potential miscalibration. This metric can be particularly useful when the use-case is high-risk and no miscalibration can be tolerated.

6.1.3 Expected Calibration Error (ECE)

The Expected Calibration Error (ECE) [Guo et al., 2017] serves as a metric to gauge the calibration quality of probabilistic predictions in classification tasks. Rather than focusing on the worst-case scenario, as with the Maximum Calibration Error

(MCE), ECE provides a weighted average of the calibration errors across all bins in a reliability diagram.

Let's formalize our notation:

- $B_1, B_2, \ldots, B_M$: The set of bins used in the reliability diagram.
- n_m: The number of samples in bin m.
- N: The total number of samples, i.e., $N = \sum_{m=1}^{M} n_m$.
- $\text{conf}(B_m)$: The average predicted probability for bin m.
- $\text{acc}(B_m)$: The accuracy of bin m.

The ECE is defined as:

$$ECE = \sum_{m=1}^{M} \frac{n_m}{N} |\text{conf}(B_m) - \text{acc}(B_m)| \tag{6.4}$$

Interpretation of ECE:

The ECE evaluates the average calibration error across all bins, weighted by the number of samples in each bin. This weighting ensures that bins with a higher number of samples contribute more significantly to the final score, highlighting the importance of regions with denser predictions.

An ECE value close to 0 indicates that, on average, the predicted probabilities of the model are well-calibrated with the observed outcomes. On the other hand, a higher ECE signifies that there's a notable discrepancy between the model's predicted probabilities and the observed frequencies. Hence, models with lower ECE values are generally deemed to be better calibrated.

However, while interpreting ECE, it's crucial to consider the binning strategy employed. The granularity and selection of bin thresholds can influence the computed ECE value, potentially leading to different interpretations of model calibration.

6.2 Experiments

We would like to build some experimental results to establish how well PWFN and the variations we have explored fare when comparing their uncertainty metrics of other methods.

One element of the method that we believe becomes more important here is the σ reassignment strategy which says, what do we do with the values for weights which have been clustered at the same time to the same mean value.

Recall that our best results with PWFN had the σ reassignment strategy where we used the mean standard deviation of the weights moved to the cluster center as the new σ value for all weights assigned to this cluster center value.

There is an open question as to whether this strategy is useful for uncertainty estimation or whether there is an alternative. It seems that there is a possibility that the 'set to mean std of weights' is not optimal. An interesting observation some weights may have had large σ values before clustering but if it is the case that they are clustered with a group that is tightly packed around the cluster center assigned, then they would be assigned a small σ value. That is σ's that we're used in Equation 4.10 to apply clustering learned during backpropagation are shifted to mean be the standard deviation of the weights that have been assigned to the same cluster value - which might have been a large travel.This works well from the perspective of clustering, but it is still optimal for the networks used as a BNN?

To test if this is indeed a problem, we further experiment with uncertainty estimations with an alternative *PWFN-sigma* where, rather than fixing σ values to the mean of the values that are assigned to the same cluster, we simply allow it to adjust itself using gradient descent.

6.2.1 Experimental Setup

We partition the ImageNet dataset into training, validation, and test sets in accordance with standard splits. We explore the uncertainty estimations for the ResNet family of models (18/34/50). For comparison, we juxtapose the performance of the PWFN, PWFN-*sigma* and COF-PWFN against three established uncertainty estimation methods: Stochastic Weight Averaging (SWA) [Maddox et al., 2019], Stochastic Weight Averaging with Gaussians (SWAG) [Maddox et al., 2019], and the uncertainty captured with an ensemble of five separately trained models (SGD). Each of these methods has been recognised for its proficiency in uncertainty estimation within neural networks.

6.3 Results

We report the Brier scores, expected confidence errors, and reliability diagrams for all the compared methods.

In Table 6.1 we present the results each method on the evaluated metrics. We see that PWFN performs well across all metrics with slightly better calibrated networks. There is clearly a benefit to setting the σ values for each of the weights distributions to be the mean of the values recently assigned to the cluster since the PWFN-*sigma* re-training does not grant the same calibration levels – despite having the similar accuracies.

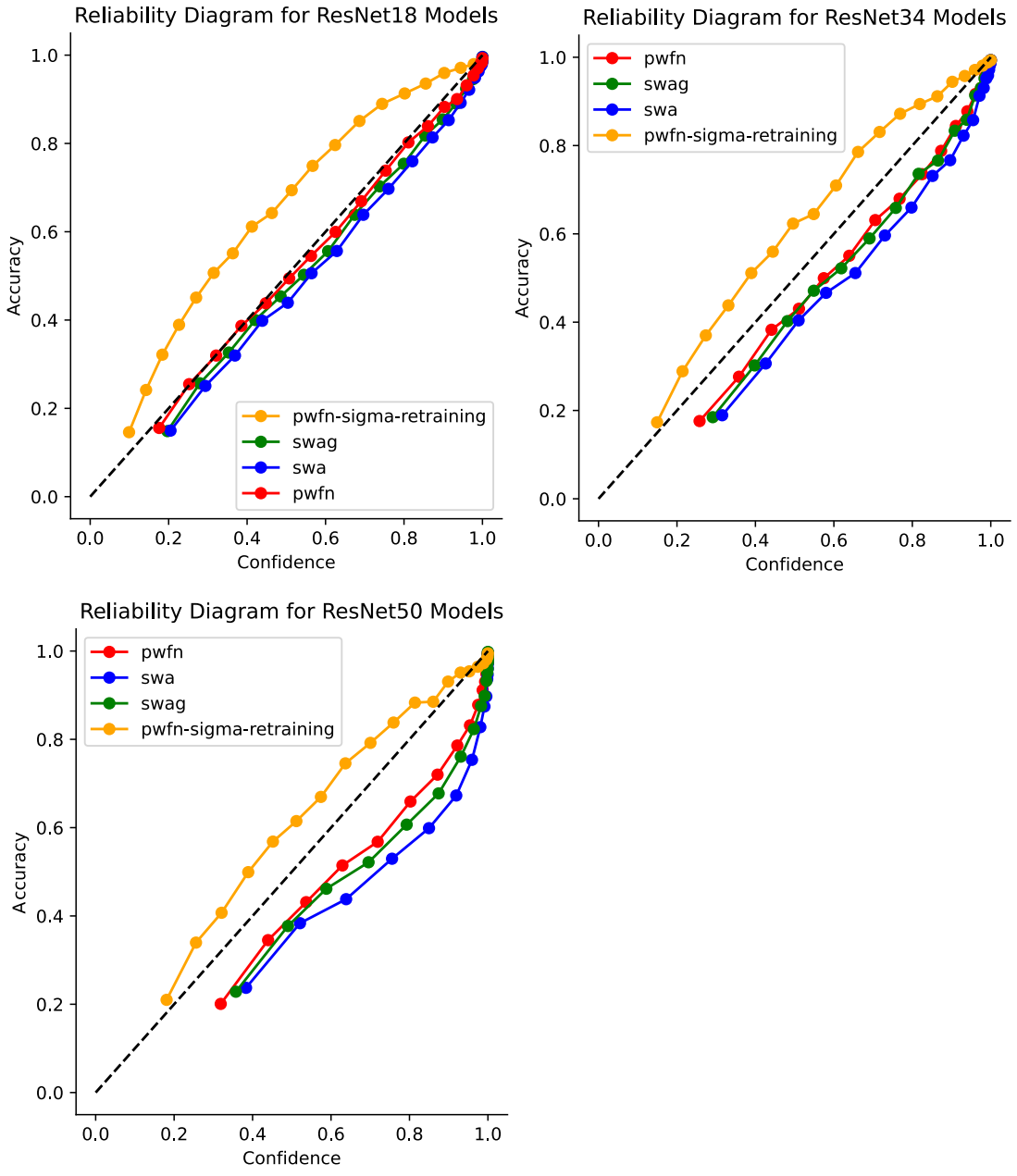


FIGURE 6.2: Here we show the reliability diagrams for the ResNet family of models tested on the ImageNet dataset. We can see that PWFN models hug the calibration line slightly closer than other methods compared demonstrating better calibration.

Model	Method	Top-1	ECE	MCE	Brier
Resnet18	SGD	70.7	0.033319	0.114677	**0.126787**
	SWAG	70.4	0.021873	0.050568	0.136844
	SWA	70.3	0.030727	0.072408	0.137099
	PWFN	70.0	**0.013016**	**0.034938**	0.142100
	PWFN-*sigma*	68.6	0.110355	0.199330	0.176671
	COF-PWFN	70.1	0.016080	0.038804	0.137921
Resnet34	SGD	74.9	0.043889	0.117737	0.139141
	SWAG	75.5	0.032063	0.106406	**0.126824**
	SWA	74.9	0.040546	0.143571	0.130269
	PWFN	74.3	**0.032669**	**0.089867**	0.133631
	PWFN-*sigma*	72.3	0.056751	0.127290	0.148675
	COF-PWFN	74.4	0.017065	0.063247	0.126940
Resnet50	SWA	79.7	0.031194	0.250969	0.118928
	SWAG	79.8	**0.030103**	0.196764	**0.113415**
	PWFN-*sigma*	77.1	0.037034	**0.116522**	0.134897
	PWFN	77.5	0.032892	0.151584	0.118640

TABLE 6.1: The uncertainty estimation results

6.3.1 Projections

Given that PWFN is able to produce good uncertainty estimations, we next would like to answer the question as to how each network sample differs in embedding space at the final layer.

One problem with using Gaussians to sample from randomly is that we are restricting the space of weight values to follow an elliptic pattern, such that none of the samples within the ellipse should change the class prediction.

The question is whether the Gaussian sampling has the same effect in embedding space. To explore this, we employ a series of projections - Principal Component Analysis (PCA) [Wold et al., 1987], Uniform Manifold Approximation and Projection (UMAP) [McInnes et al., 2018], and t-Distributed Stochastic Neighbor Embedding (t-SNE) [Van der Maaten and Hinton, 2008] - to visualise the logit (Figure 6.4) and penultimate activation layers (Figure 6.3). In these projections, we color-code various image samples, using consistent colors within a subplot to represent the outputs for different weight samples for the each input image. Squares and circles respectively mark incorrect and correct predictions, offering an immediate visual cue of the model's performance.

Upon examination, several patterns emerge. First, we observe that t-SNE and UMAP projections yield similar distributions of weight samples, suggesting a relatively uniform transformation through these techniques. However, a more

FIGURE 6.3: Penultimate Activation Layer Projection in ResNet-18 with PWFN (PCA, UMAP, t-SNE): This projection shows the penultimate activation layer of a ResNet-18 model using the PWFN method. Similar to Figure 1, each colour signifies a single image input, with five samples for each, and the projections are arranged from left to right: PCA, UMAP, and t-SNE. This arrangement allows for direct comparison of how each dimensional reduction technique affects the visualisation of weight sample variability and the corresponding predictions.

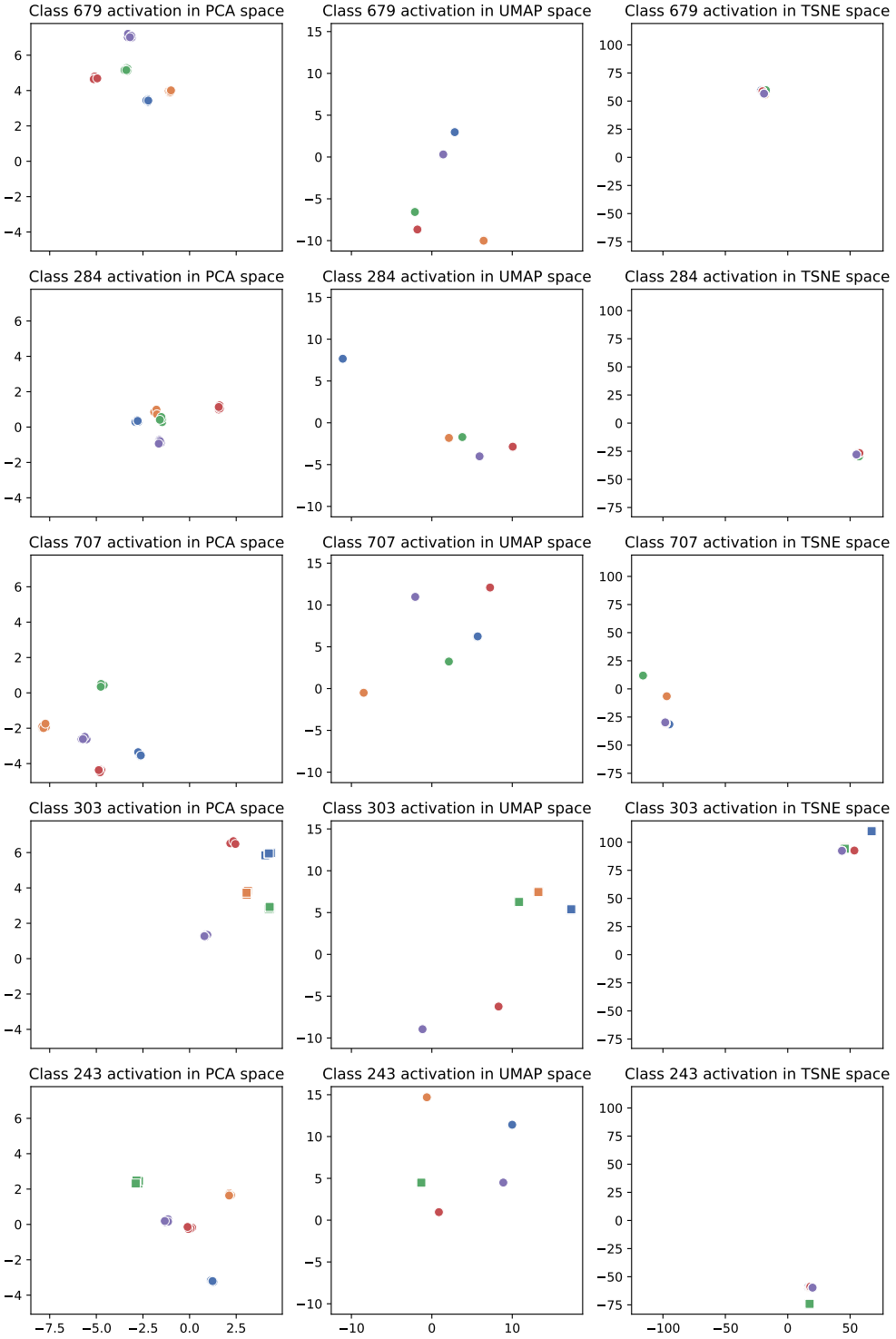
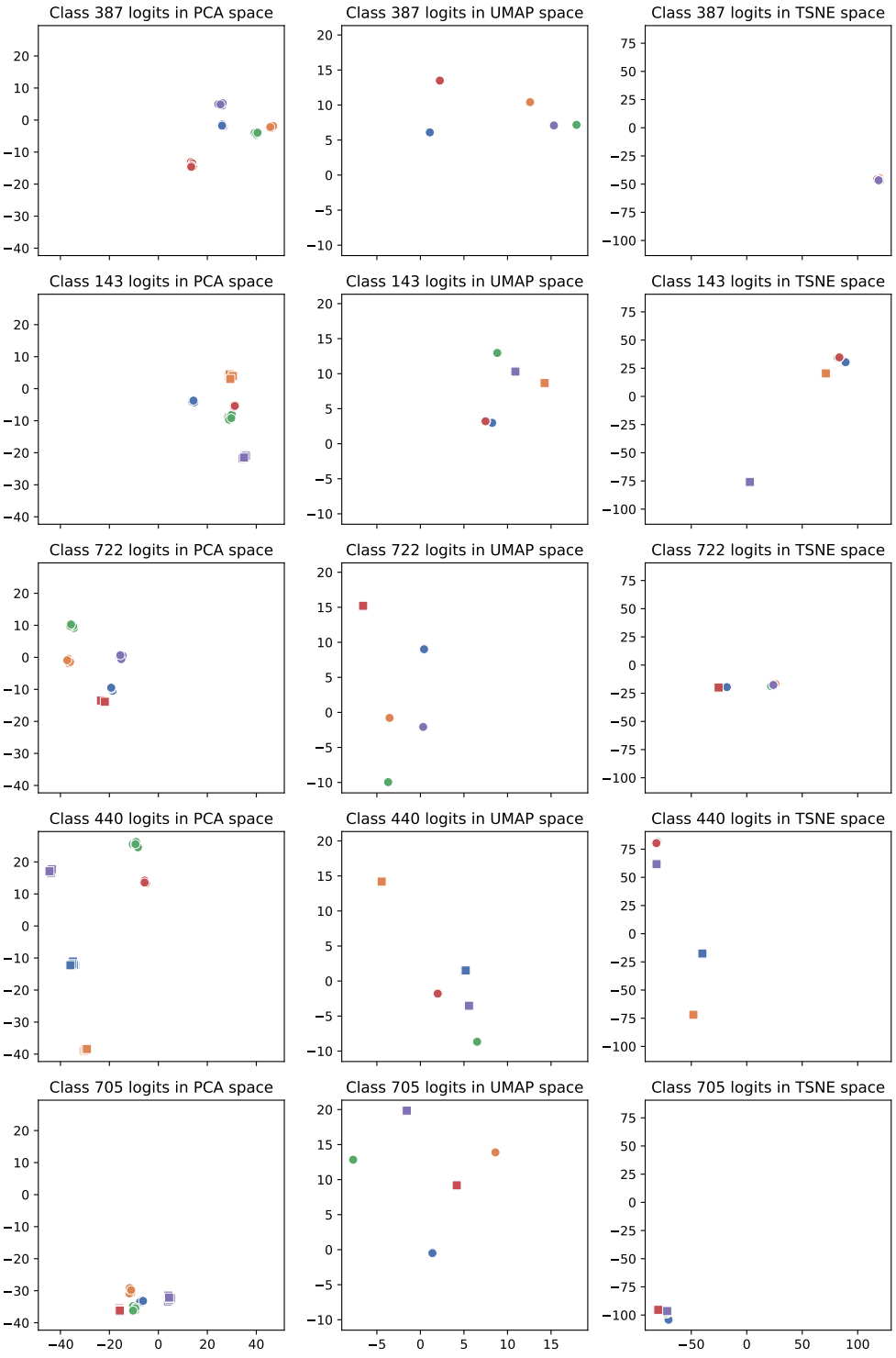


FIGURE 6.4: Comparative Analysis of Logit Projection via PCA, UMAP, and t-SNE in ResNet-18 with PWFN: This figure displays logit projections for a ResNet-18 model trained with PWFN, using PCA (left), UMAP (middle), and t-SNE (right). Each colour represents a unique image input, with five instances per colour, denoting the five weight samples per input. Crosses and circles indicate incorrect and correct predictions, respectively, illustrating the impact of weight sample variability on the predictive accuracy.



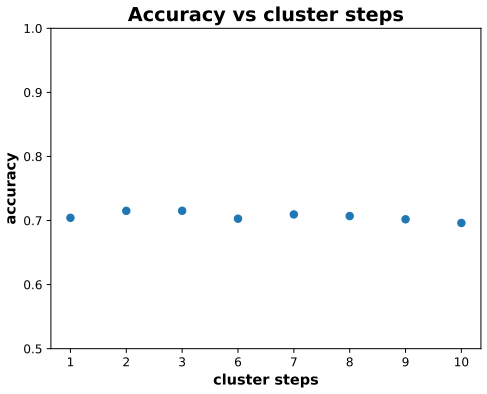


FIGURE 6.5: We can see here that with each cluster step (where we increase the number of weights that are clustered according to 4.8 (left)) that the model accuracy is maintained.

pronounced variation is discernible in the PCA projection, particularly within the two principal components examined.

Interestingly, despite sampling across all layers, the overall variation between weight samples in both projected logits and activations remains moderate. There are much larger distances between different image inputs of the same class and different samples of the same image, suggesting that in loss-space, we are finding distinct valleys and exploring to the edge of where the loss would begin to increase. One interesting example is in Figure 6.3 class 284 (second row, far left), where we can see the yellow and green different image samples are projected close to each other and have overlapping outputs given random network sampling. It is difficult with just two dimensions to be sure, and further investigation is needed, but it implies that it is possible to learn a weights' distribution where sampling of different images would project into the same loss valley.

Bringing together these results and the uncertainty estimation results implies that excessive variability in weight samples could potentially lead to erratic class predictions as they traverse the logit space and that when projected down onto just two dimensions, the weight sampling has only a small effect in changing the activation/logit locations. The relatively stable projection patterns we observe suggest a form of inherent equilibrium in the network, where the compounded effects of weight variation either neutralize each other in the deeper layers or do not significantly sway the overall prediction outcomes.

6.4 Compression Vs Calibration

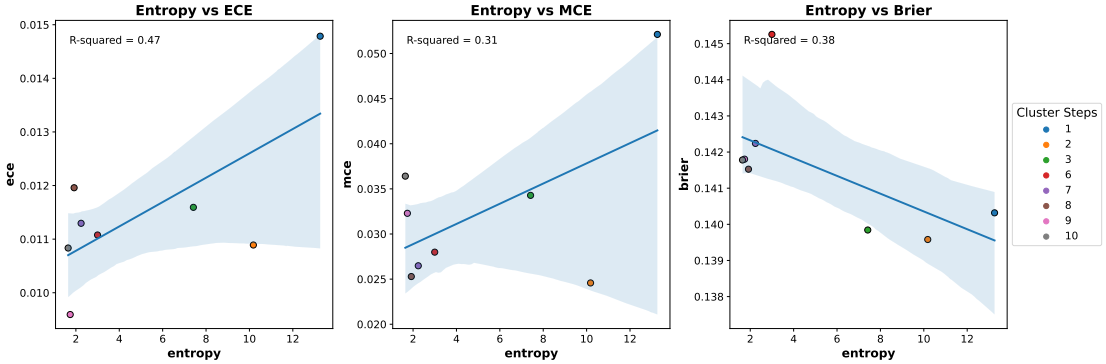


FIGURE 6.6: Relationship between entropy and calibration metrics (ECE, MCE, and Brier) for different numbers of clustering steps. The scatter plots show the correlation between entropy and each calibration metric, with the line of best fit indicating the overall trend. The R-squared value is provided for each plot, quantifying the strength of the linear relationship. Higher entropy values tend to correspond to better calibration (lower ECE, MCE, and Brier scores), suggesting that models with higher entropy are better calibrated.

Next we look to answer the question does compression help with calibration. That is, are networks that undergo the PWFN quantisation steps better calibrated than those that don't.

This question is one which is answerable under the PWFN training setting. This is because we have a set-up with maintains accuracy whilst varying compression (see Figure 6.5). In order to test the hypothesis that compression results in better calibration we look at the ECE, MCE and Brier scores with each clustering stage of PWFN.

In Figure 6.6, we present these results. We observe a clear pattern as we increase the clustering steps: the entropy decreases, and both the ECE and MCE reduce, suggesting better calibration. Interestingly, the Brier score exhibits the opposite behavior, increasing with more clustering steps. This suggests that while the model's calibration improves, as evidenced by the decreasing ECE and MCE, the refinement aspect may be compromised.

Refinement, in this context, refers to the model's ability to assign high probabilities to correct classes and low probabilities to incorrect classes. A well-refined model should be confident in its predictions for the correct classes and uncertain about the incorrect ones. The increasing Brier score indicates that although the predicted probabilities align better with the observed accuracies (i.e., improved calibration), the model might be assigning lower probabilities to the correct classes or higher probabilities to the incorrect classes.

This trade-off between calibration and refinement highlights the importance of considering multiple evaluation metrics when assessing a model's performance.

While the clustering steps in PWFN improve calibration, it is crucial to strike a balance between compression and maintaining the model's ability to make accurate and confident predictions. Further analysis of the model's confidence scores and their relationship to the correctness of predictions can provide insights into the impact of clustering on the model's refinement and overall predictive accuracy.

The Brier score's sensitivity to class imbalance and equal emphasis on refinement may not fully align with the primary objective of investigating the impact of clustering on calibration in this particular setting. Therefore, the Expected Calibration Error and Maximum Calibration Error serve as more direct and interpretable metrics for assessing the calibration performance of PWFN, while the Brier score provides a complementary perspective on the model's overall performance.

Let us now move to explore a related topic of how well the uncertainy estimations in PWFN can help us identify out-of-distribution (OOD) datapoints.

6.5 Out-of-distribution

In this section, we conduct a preliminary investigation to evaluate the performance of PWFN in identifying samples that deviate from the training distribution. While we use the term "out-of-distribution" (OOD) in our experiments and figures, what we're examining is more precisely characterised as outlier recognition within a controlled experimental setting. This investigation aims to elucidate the model's ability to detect data points that fall outside the scope of its training dataset classes, a capability critical for real-world deployment.

Before we proceed to the experiments, let's briefly explore the rationale behind studying this detection capability.

6.5.1 Understanding the Need for OOD Metrics

In real-world machine learning systems, it is often the case that data fed in to a model to be classified lies outside the given set of classification classes that a model has been trained on. This datapoint - referred to as OOD will need to be handled gracefully by the system [Berend et al., 2020, Yang et al., 2021].

The robustness and reliability of a model in practical applications are significantly determined by its ability to handle this OOD data. This ability is for several reasons:

- **Adapting Behaviour**: When encountering OOD data, we might want to change the model's behaviour to ensure safety, reliability, and user satisfaction, such as falling back to conservative actions or alerting human operators in critical scenarios [Amodei et al., 2016, Hendrycks et al., 2021].
- **Detecting Anomalies and Potential Threats**: OOD data may indicate anomalies, malicious attacks, or unexpected situations. Detecting such instances is crucial for maintaining the system's integrity and security [Ruff et al., 2021, Carlini and Wagner, 2017].
- **Model Calibration and Uncertainty Estimation**: OOD metrics provide insights into the model's calibration and can help estimate its uncertainty in predictions, preventing blind reliance on potentially inappropriate outputs [Guo et al., 2017, Ovadia et al., 2019].
- **Identifying Limitations and Guiding Data Collection**: Analysing encountered OOD data helps identify the model's limitations and guides future data collection efforts to improve coverage and robustness [Bulusu et al., 2020, Pimentel et al., 2014].
- **Distinguishing Error Types for Debugging and Improvement**: Secondary to the classification task, differentiating between in-distribution and OOD errors provides valuable information for debugging and model improvement, pointing to issues with the model itself or limitations in training data coverage [Hecker et al., 2018, Kendall and Gal, 2017].
- **Enhancing Robustness through Targeted Retraining**: Identifying OOD errors allows for the selection of the most relevant and effective data points for robustness enhancement through targeted retraining, improving the model's performance in real-world scenarios [Hendrycks et al., 2019, Zheng et al., 2020].

6.5.2 Studying OOD Metrics within PWFN

Aside from the general motivations for studying the OOD setting, we believe that treating PWFN as a Bayesian approach makes it potentially more fruitful in detecting OOD samples without any additional tricks. One would hope that the sampling procedure will give more variation of outputs for input classes which are unseen in the training set.

Additionally, understanding the performance of compressed models (a notable advantage of PWFN) on OOD data is essential for ensuring that efficiency gains do not compromise robustness.

With these considerations, let us next move on to describing the experimentation undertaken to evaluate OOD on PWFN.

6.5.3 OOD Experimentation with PWFN

For the OOD experiments, we look to the CIFAR-10 dataset for simplicity, segregating it into in-distribution (ID) and OOD sets. Five classes are designated for ID training, and the remaining five classes are reserved for evaluating the model's OOD detection capabilities.

We next apply PWFN for training on the ID data where we expect to reach high accuracies and low uncertainty of predictions and then evaluate how this changes when we test on OOD datapoints.

6.5.3.1 Entropy of Probabilities

To better understand the distinction between ID and OOD samples, we calculate the entropy of class probabilities for a range of sample numbers (1, 5, 10, and 20) during the testing phase on both OOD and ID data. In this experiment we take n samples (n=1,5,10,20) of the weights and calculate the entropy of the probabilities for each data input. The idea is to explore different sample sizes allowing us to investigate the impact of the number of samples on the model's ability to distinguish between ID and OOD data points. This metric provides valuable insights into the model's performance, with a higher difference between OOD and ID data points indicating more effective OOD detection. The core idea behind this approach is that it is a simple metric to compute, and we expect to observe a significant variation between ID and OOD test samples, especially as the number of samples increases.

6.5.3.2 Standard Deviation Thresholding

In addition to the entropy analysis, we explore the viability of an approach to automate the process of determining when a sample is OOD. To achieve this, we threshold the entropy values at various levels based on standard deviations above the mean entropy of all test samples. This analysis helps us understand the model's sensitivity and response to OOD data under varied conditions and demonstrates how these metrics could be applied in practice.

Visualisation and Interpretation To supplement our analysis, we present visual representations of the results:

- **Accuracy Plot**: Figure 6.8 illustrates the model's accuracy in identifying ID and OOD classes across different entropy thresholds.
- **Entropy Distribution Plots**: Figure 6.7 visually depicts the distribution of entropy values for ID and OOD classes, highlighting the differences between the baseline model and the PWFN models with varying numbers of samples.

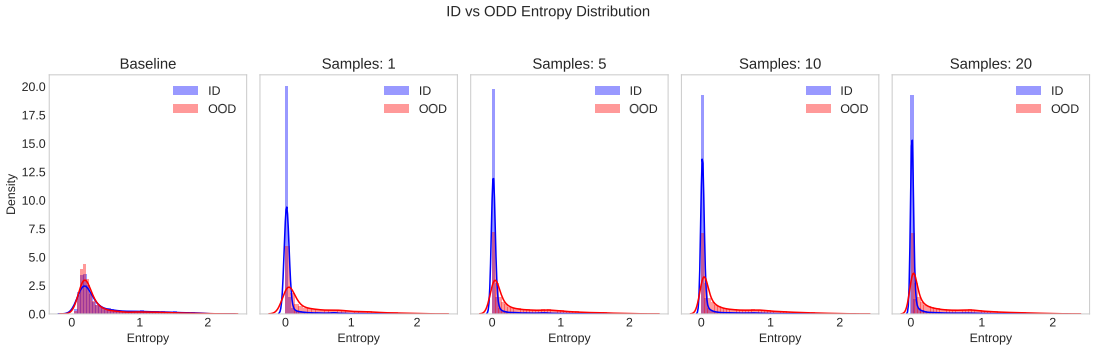


FIGURE 6.7: Here we show how the distribution of soft-max entropy values changes between ID and OOD samples in the reduced-CIFAR-10 experiments. We can see that the baseline model (left) doesn't have much difference between the two whereas the PWFN models at varying number of samples has a much more differentiation between the two.

6.5.4 Experiment Results and Discussion

Let's start by examining how the distributions of ID and OOD samples differ, keeping in mind that a *good* model would exhibit higher softmax entropies for OOD samples and much lower entropies for ID samples. In Figure 6.7, we explore the distribution difference between entropies for a baseline (standard single SGD run) and PWFN models taking 1, 5, 10, and 20 samples. It's crucial to note that each of these models has comparable accuracy for the ID classes, yet we observe a substantial difference when comparing the softmax entropies of ID and OOD samples. Even with a single sample, the ability to distinguish between ID and OOD is much clearer than with the baseline model. As we increase the number of samples, the ID distribution becomes even more concentrated towards lower entropy values, making the OOD samples even more distinguishable.

Let's examine this concept further. Suppose we wanted a simple measure to determine whether a sample is ID or OOD. One approach could be to sample the weights n times and calculate the entropy of the model's softmax output for a given input.

For $n = 1$, we calculate the entropy $H(x)$ directly from the softmax output probabilities. For $n > 1$, we take n samples from the weights, calculate the mean softmax probabilities for the input, and then compute the entropy from the averaged probabilities.

Given the entropy $H_n(x)$ calculated for n samples, we can compare it to the mean entropy μ_H over all data samples or from a training calibration set (which are all ID). The z-score z_x, representing how many standard deviations $H_n(x)$ is from μ_H, is defined as:

$$z_x = \frac{H_n(x) - \mu_H}{\sigma_H}$$

where σ_H is the standard deviation of the entropy across the same set of samples. Using this z-score, we can set a threshold τ, such that if $z_x > \tau$, we classify the sample as OOD:

$$\text{Classify as OOD if } z_x > \tau$$

We explore a range of τ values corresponding to 0, 0.25, 0.5, 0.75, 1, 1.25, 1.5, 1.75, and 2 standard deviations, and analyse the percentage of ID and OOD samples that exceed this threshold. Based on our findings from examining the entropy distributions, we expect the PWFN models to have a higher percentage of OOD samples and a lower percentage of ID samples above the threshold compared to the baseline model.

This formulation helps formalise the distinction between ID and OOD samples using the average entropy over multiple weight samples and the deviation from the mean entropy.

Figure 6.8 illustrates the percentage of OOD and ID samples that fall above a certain number of standard deviations from the mean entropy. The number next to ID/OOD indicates the number of samples used in the PWFN model. We can see that PWFN, with any number of samples $n \geq 1$, results in the highest percentage of OOD samples exceeding the threshold. For instance, at 0 standard deviations above the mean (i.e., the mean itself), 25% of OOD samples are correctly identified, compared to less than 5% of ID samples, making it a far more effective classifier than the non-Bayesian baseline, which only detects 15% of OOD samples.

These findings underscore the effectiveness of the PWFN approach in distinguishing between ID and OOD samples, even with a small number of weight samples. The greater separation between the entropy distributions of ID and OOD samples, alongside the higher percentage of OOD samples above the entropy threshold, highlights the potential of this method for reliable OOD detection in real-world applications. Using the PWFN approach, practitioners can build more robust and trustworthy systems capable of effectively managing OOD data, reducing the likelihood of unexpected failures or erroneous predictions.

Although this metric is not perfect—capturing only 25% of OOD examples and misclassifying approximately 5% of ID samples as OOD—it still demonstrates the potential of using the additional variance signal from PWFN for OOD detection. Further refinement is required, but these results are promising.

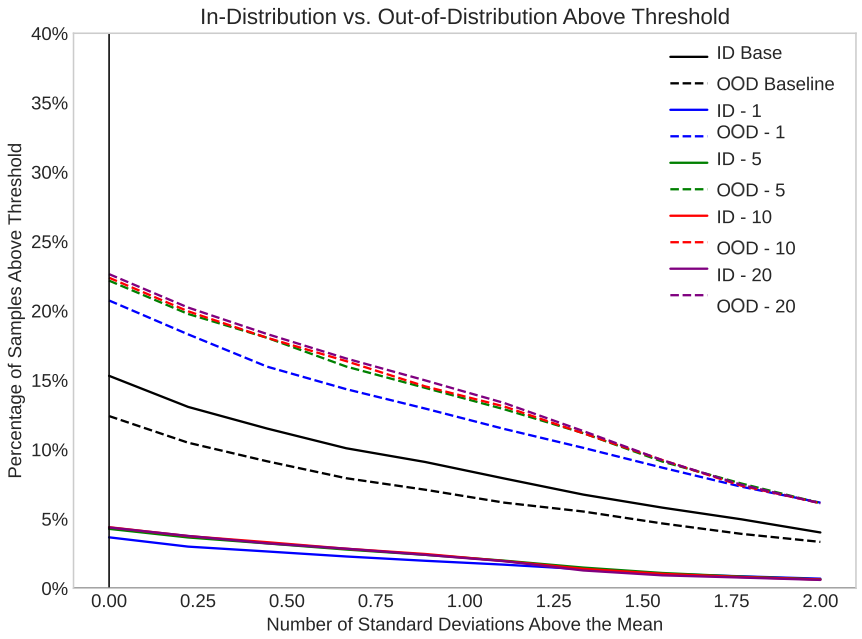


FIGURE 6.8: Here we show what percentage of OOD and ID samples are above x number of standard deviations above the mean. The number next to ID/OOD signifies how many samples were used in the PWFN case. We can see that using WFN with any number ≥ 1 samples gives us the highest percentage of detected OOD samples. Taking 0 std above the mean (i.e., the mean itself) for example, we can see that 25% of OOD samples versus less than 5% ID samples exceed the threshold, making it a much better classifier than if we were to use the non-Bayesian baseline, where only 15% of OOD samples would be detected.

6.6 Conclusion

We conclude by noting that both the OOD and uncertainty estimation experiments demonstrate PWFN is a step towards the goal of having well calibrated models. We have seen that the interaction between compression and calibration is one of benefit. That compressed models tend to give better calibration. The set of experiments and ablation studies conducted in this chapter provide a glimpse into the relationship of three hitherto separate fields of inquiry, PWFN gives us better calibration, generalisation (to the test set) and compression. We are now in a good place to look back at this thesis and to start to link the threads together.

Chapter 7

Discussion and Further Analysis

7.1 The Thesis Contributions

In this thesis, we presented a sequence of algorithmic techniques for neural network compression, each building upon the insights and findings of its predecessor. Our primary objective was to reduce data-movement costs and enable efficient inference on specialised hardware. We identified that data-movement costs dominate energy expenditure in deep learning inference (Chapter 2), motivating the exploration of techniques that reduce the number of unique parameters and weight-space entropies.

We began with Weight Fixing Networks (WFN) in Chapter 3, introducing an iterative cluster-train algorithm that employed techniques such as iterative quantisation, a whole-network codebook, a focus on powers-of-two clusters, and a relative distance movement threshold. WFN effectively reduced the number of unique weights and weight-space entropy but relied on a fixed schedule for clustering and did not leverage weight-specific information for quantisation decisions.

In Chapter 4, we introduced Probabilistic Weight Fixing Networks (PWFN), building on the lessons learned from WFN. PWFN incorporated a probabilistic framework based on Bayesian neural networks and variational relaxation, leveraging learned weight uncertainties to make context-aware quantisation decisions. PWFN achieved state-of-the-art results on ImageNet with ResNet and transformer models, demonstrating the power of learning noise-resilient, highly compressible weight configurations. However, PWFN still relied on a predefined schedule for clustering stages.

To address the limitations of the fixed clustering schedule in PWFN, Chapter 5 introduced Cluster-On-the-Fly PWFN (COF-PWFN). COF-PWFN integrated the clustering process into the training phase itself, making it more dynamic and adaptive. While COF-PWFN showed promising results in terms of accuracy, it

also highlighted a potential trade-off between accuracy and compression that warrants further exploration.

Additionally, we explored the potential benefits gained from the probabilistic viewpoint for uncertainty estimation in Chapter 6. By leveraging the probabilistic nature of PWFN and COF-PWFN, we demonstrated their capabilities in providing reliable uncertainty estimates and out-of-distribution detection.

7.2 Implications and Future Directions

The techniques and insights presented in this thesis have implications for the development of efficient and scalable deep learning systems. The proposed algorithmic approaches—WFN, PWFN, and COF-PWFN—demonstrated the potential for reducing the number of unique weights and weight-space entropy in neural networks while maintaining performance.

7.2.1 Hardware Constraints and Data Movement Costs

One implication of this work is the importance of considering hardware constraints and data movement costs when designing algorithmic approaches for neural network compression. We highlighted the need for collaboration between algorithm developers and hardware designers to leverage the benefits of these compression techniques on specialised hardware platforms. This collaboration can lead to the development of bespoke hardware implementations tailored to the specific characteristics of the compressed models, potentially unlocking further efficiency gains. As new algorithmic architectures are developed, there is often a shift in hardware to support it. Hardware that works well for inference of generative adversarial networks is not as suited to running diffusion models, but the principle of minimising data-movement is universal as a result of the fundamentals of the physics of computational chips. One could argue that the work presented in this thesis is not algorithm-hardware co-design but is instead adapting current algorithms such that they are better suited to hardware strengths and limitations. We accept the premise that more work is needed to bring algorithmic development even closer to the hardware and point to research teams out of FireworksAI [Fireworks AI, 2024], Groq [Abts et al., 2022], and of course Nvidia [Liu et al., 2023], which are doing just that.

7.2.2 Hardware Realisation of WFN

Our WFN approach is well-suited for efficient hardware implementation due to its emphasis on minimising unique weight values and weight-space entropy across the entire network.

7.2.2.1 Codebook-Based Architecture

A hardware architecture implementing our methods would use a compact, shared codebook stored in fast SRAM accessible to processing elements. For networks compressed with our approach, where millions of parameters are represented by as few as 100-200 unique values, this architecture reduces memory traffic between off-chip storage and on-chip compute units.

Similar codebook-based architectures exist in hardware accelerators such as the UNPU [Lee et al., 2018], which demonstrated energy reductions of 23.1% to 53.6% for different weight precision levels compared to conventional fixed-point arrays. The EIE accelerator [Han et al., 2016a] also showed that index-based representations with a small codebook achieve substantial energy savings.

7.2.2.2 Huffman Encoding for Weight Indices

To further optimise memory usage, Huffman encoding for the weight indices would be highly effective. Given the highly skewed weight distribution produced by WFN (as shown in our results), Huffman coding works particularly well:

1. **Efficient Huffman Decoder**: Research by Tian et al. [Tian et al., 2021] demonstrated that modern GPU-based Huffman decoders can achieve throughput rates exceeding 300 GB/s on NVIDIA V100 GPUs. For inference acceleration, Hashemian [Hashemian, 1994] showed memory-efficient hardware implementation of Huffman decoding that reduced memory requirements by nearly 67× (from 8192 words to 122 words) for codebooks with codewords up to 13 bits.
2. **Compression Benefits**: The EIE accelerator [Han et al., 2016a] demonstrated that Huffman coding provided an additional 2.4× reduction in memory requirements with minimal decompression overhead, making it well-suited for our WFN approach which generates highly skewed weight distributions.

7.2.2.3 Power-of-Two Optimisations

The skewed distribution of weights in our WFN models—where over 75% are power-of-two values—enables specific hardware optimisations. Multiplication operations for power-of-two weights can be replaced with simpler bit-shift operations, reducing computational complexity and energy consumption. This can be implemented using specialised datapaths that detect power-of-two values and route the computation through bit-shifters instead of multipliers.

7.2.2.4 Whole-Network Shared Codebook

Our whole-network shared codebook approach differs from typical layer-wise codebooks used in many quantisation method since the unified codebook can be loaded once at the beginning of inference and cached throughout the entire forward pass, as demonstrated in Eyeriss v2 [Chen et al., 2019], which showed that optimising data reuse across the entire network can reduce energy consumption by up to 2.5×.

7.2.2.5 Memory-Efficient Implementation

Drawing from Hashemian's work [Hashemian, 1994], our design could implement a memory-efficient Huffman decoding structure based on code-bit clustering. This approach would allow the decoder to recognise variable-length codes embedded in a continuous stream of bits. By using a lookup table (LUT) with our small codebook (100-200 entries), we can achieve significant memory savings compared to traditional implementations requiring 2^k memory locations (where k is the maximum codeword length).

The design would consist of three main components:

1. A buffer to hold incoming weight indices
2. A LUT address generator that extracts variable-length Huffman codes
3. A source code generator that outputs the actual weight values

In conclusion, our WFN approach could be efficiently implemented in hardware using a combination of codebook-based architecture, Huffman encoding, and power-of-two optimisations.

7.2.3 Uncertainty Estimation in Safety-Critical Applications

In this work we also introduced the uncertainty estimation capabilities demonstrated by PWFN and COF-PWFN have implications for the development of reliable and interpretable deep learning systems. The idea is that these techniques can be applied to safety-critical applications, such as autonomous driving and medical diagnosis, where the reliability and calibration of model predictions are crucial. The ability to quantify and communicate the uncertainty associated with model predictions can enhance trust in deep learning systems and facilitate their integration into decision-making processes. We have seen that using PWFN in particular, we were able to use probabilistic uncertainty estimations to calibrate models such that low probability predictions do indeed match up with low probability of correctness. As far as we know, this is the first work to achieve large-scale training of probabilistic uncertainty in deep learning

through the variational network route. The simplicity of the approach (maximising noise resilience) means this type of approach is no longer limited to toy problems.

7.2.4 Exploring Accuracy-Compression Trade-offs

This thesis also opens up several avenues for future research. One direction is to further investigate the accuracy-compression trade-offs in COF-PWFN and develop techniques to optimise this balance.

Our work supports the ideas proposed in the minimal description length, that "compression is comprehension". We found that PWFN networks were better calibrated as they were further compressed (up to a limit) suggesting that the accuracies achieved in training can be coupled with a compression stage to further refine the model. This is well supported across the results found in quantisation works which have shown time and again, that an additional quantisation step – particularly when guided by optimisation – can increase downstream task generalisability [Yuhang Li, Xin Dong, 2020, Subia-Waud and Dasmahapatra, 2022, Fan et al., 2021, Lee et al., 2021, Jin et al., 2020]. So far, this finding has not translated into large language models, where the move from 8-bit down to 4-bit is coupled with a slight drop in performance [Lin et al., 2024, Frantar et al., 2022]. It is an open question as to whether this is due to the sheer size of the training set – Llama-3 was trained with over 1 trillion tokens whereas vision models are trained with orders of magnitude fewer – or whether some important component is missing in attempts to quantise, training with gradient descent. Post-training quantisation approaches for vision models are similarly coupled with small drops in performance. A challenge in exploring the accuracy-compression trade-off for these larger models lies in the training costs which currently restrict large-scale training to a select few organisations. There is a risk-reward consideration which may limit the extent to which compression within training is researched but we pose from the results in this work and others, that it could be a fruitful avenue of exploration.

7.2.5 Integration with Other Approaches

In this work, we have attempted to isolate the effect of compression through quantisation/weight-sharing alone. Other works, most notably Deep Compression [Han et al., 2016b], have shown that other compression techniques such as pruning can be used in tandem with quantisation in a multi-stage process. Although pruning is somewhat already present (0 was always the modal cluster in PWFN and WFN) [Han et al., 2015a], other techniques such as neural architecture search [Zoph and Le, 2016], low-rank approximation [Denton et al., 2014], and

entire channel pruning [Li et al., 2016] are potentially complementary techniques which may be combined with PWFN/WFN in order to further compress networks. This integration could lead to the development of comprehensive frameworks for neural network compression that can automatically adapt to the specific requirements of different tasks and hardware platforms [He et al., 2018].

7.2.6 Adapting to Emerging Architectures

Finally, applying the proposed techniques to the realm of large language models (LLMs) in the evolving landscape of artificial intelligence is an important area for future exploration. As the scale and complexity of these models continue to grow, the need for efficient compression and inference techniques becomes increasingly pressing.

LLMs are already typically on the scale of billions of parameters. A recent open-source release of Llama-3.1 [Meta AI, 2024] for example, has over 400 billion parameters, far more than is viable to run on any consumer hardware. Quantisation has already become a core component of enabling access to local inference of LLMs. New techniques such as activation aware quantisation [Lin et al., 2024] and GPTQ [Frantar et al., 2022], both post-training quantisation techniques, are already very popular. Llama-3 4-bit by unsloth [UnslothAI, 2024] is, as of writing this, the 4th most downloaded model on Huggingface.

As was the case with vision models, these post-quantisation methods for LLMs result in accuracy drops - depending on how far the bit-width is pushed [Chavan et al., 2024]. New approaches have begun to emerge with a focus on quantisation aware training [Ma et al., 2024]. As with vision, these approaches trade off compression and accuracy against granularity with separate codebooks.

The K,V re-use seen in transformers (each next token prediction re-uses the same weights and activations of the previous token prediction) fits well within the ideas outlined in this thesis. Our hope is that the insights we have gained will be applied to LLMs in the future. There are additional computation costs (the extra σ values to be trained) which would make it out-of-reach for current consumer devices to utilise but should this hurdle be overcome, cutting down the number of unique values across a network and keeping the weights close to the computational units would be a huge boon to faster inference speeds of LLMs.

Indeed, companies like Groq has focused on this principle to incredible effect. They have developed language processing units (LPUs) [Groq, 2024] which compile a network down onto an accelerator where all of the weights are right next to a processing element and the data feed through with tokens per second speeds far greater than any of the competition.

It is our hope that the insights and principles introduced in this thesis could aid in the development of specialised compression techniques that can handle the unique challenges posed by these large-scale models.

7.3 Conclusion

This thesis has presented a comprehensive investigation into neural network compression techniques, driven by the imperative to minimise data-movement costs in deep learning inference. Our research has yielded a series of algorithmic approaches—Weight Fixing Networks (WFN), Probabilistic Weight Fixing Networks (PWFN), and Cluster-On-the-Fly PWFN (COF-PWFN)—each advancing the state of the art in efficient inference.

Our work was motivated by the observation that data-movement costs dominate energy expenditure in deep learning inference, as established in Chapter 2. This insight guided our efforts towards reducing the number of unique parameters and weight-space entropies, a strategy that has proven effective in enhancing inference efficiency.

The progression of our research, from WFN through to COF-PWFN, demonstrates a systematic approach to addressing the challenges of neural network compression. WFN introduced an iterative cluster-train algorithm, while PWFN leveraged Bayesian neural networks and variational relaxation for context-aware quantisation decisions. COF-PWFN further refined this approach by dynamically integrating the clustering process into the training phase.

A significant outcome of our work, particularly with PWFN and COF-PWFN, is the improvement in model calibration and uncertainty estimation. These capabilities extend the potential applications of our compressed models to domains where reliable uncertainty quantification is crucial.

Our research has highlighted several key areas for future investigation:

- **Hardware-Algorithm Integration**: The findings underscore the importance of aligning algorithmic approaches with hardware constraints. Future research should focus on closer collaboration between algorithm developers and hardware designers to fully exploit the benefits of these compression techniques.
- **Accuracy-Compression Trade-offs**: The relationship between compression and model calibration, particularly in PWFN, merits further study. Investigating the potential for an optimal balance between compression, computational efficiency, and model performance could yield valuable insights.

- **Synergy with Other Compression Techniques**: While this thesis focused primarily on quantisation and weight-sharing, integrating our methods with techniques such as pruning, neural architecture search, and low-rank approximation could lead to more comprehensive compression frameworks.
- **Application to Large Language Models**: Adapting these techniques to the scale of Large Language Models presents both significant challenges and opportunities. This area of research could potentially address the growing computational demands of these massive models.
- **Practical Implementation of Uncertainty Estimation**: Translating the uncertainty estimation capabilities of our probabilistic approaches into practical applications, particularly in safety-critical domains, remains an important area for future work.

In conclusion, this thesis has contributed to the ongoing research in efficient and reliable deep learning systems. By grounding our approach in the fundamental principles of computational efficiency and data movement minimisation, we have developed techniques that not only compress neural networks but also enhance their interpretability and reliability.

The challenges that lie ahead—from scaling these techniques to massive language models to integrating them with specialised hardware—present significant opportunities for future research. As the scale and complexity of deep learning models continue to grow, the need for efficient, reliable, and interpretable systems becomes increasingly critical. The principles and techniques introduced in this thesis provide a foundation for addressing these challenges, potentially leading to more scalable and efficient deep learning systems in the future.

Bibliography

D. Abts, J. Kim, G. Kimmell, M. Boyd, K. Kang, S. Parmar, A. Ling, A. Bitar, I. Ahmed, and J. Ross. The groq software-defined scale-out tensor streaming multiprocessor: From chips-to-systems architectural overview. In *2022 IEEE Hot Chips 34 Symposium (HCS)*, pages 1–69. IEEE Computer Society, 2022.

D. Amodei, C. Olah, J. Steinhardt, P. Christiano, J. Schulman, and D. Mané. Concrete problems in ai safety. *arXiv preprint arXiv:1606.06565*, 2016.

C. Banbury, C. Zhou, I. Fedorov, R. Matas, U. Thakker, D. Gope, V. Janapa Reddi, M. Mattina, and P. Whatmough. Micronets: Neural network architectures for deploying tinyml applications on commodity microcontrollers. *Proceedings of Machine Learning and Systems*, 3:517–532, 2021.

E. Begoli, T. Bhattacharya, and D. Kusnezov. The need for uncertainty quantification in machine-assisted medical decision making. *Nature Machine Intelligence*, 1(1):20–23, 2019.

D. Berend, X. Xie, L. Ma, L. Zhou, Y. Liu, C. Xu, and J. Zhao. Cats are not fish: Deep learning testing calls for out-of-distribution awareness. In *Proceedings of the 35th IEEE/ACM international conference on automated software engineering*, pages 1041–1052, 2020.

D. Blalock, J. J. G. Ortiz, J. Frankle, and J. Guttag. What is the State of Neural Network Pruning? mar 2020. URL http://arxiv.org/abs/2003.03033.

C. Blundell, J. Cornebise, K. Kavukcuoglu, and D. Wierstra. Weight uncertainty in neural networks. In *32nd Int. Conf. Mach. Learn. ICML 2015*, volume 2, pages 1613–1622, 2015. ISBN 9781510810587.

A. Boroumand, S. Ghose, Y. Kim, R. Ausavarungnirun, E. Shiu, R. Thakur, D. Kim, A. Kuusela, A. Knies, P. Ranganathan, et al. Google workloads for consumer devices: Mitigating data movement bottlenecks. In *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 316–331, 2018.

G. W. Brier. Verification of forecasts expressed in terms of probability. *Monthly weather review*, 78(1):1–3, 1950.

S. Bulusu, B. Kailkhura, B. Li, P. K. Varshney, and D. Song. Anomalous example detection in deep learning: A survey. *IEEE Access*, 8:132330–132347, 2020.

N. Carlini and D. Wagner. Towards evaluating the robustness of neural networks. In *2017 IEEE Symposium on Security and Privacy (SP)*, pages 39–57. IEEE, 2017.

L. Cavigelli, D. Gschwend, C. Mayer, S. Willi, B. Muheim, and L. Benini. Origami: A convolutional network accelerator. In *Proceedings of the 25th edition on Great Lakes Symposium on VLSI*, pages 199–204, 2015.

A. Chavan, R. Magazine, S. Kushwaha, M. Debbah, and D. Gupta. Faster and lighter llms: A survey on current challenges and way forward. *arXiv preprint arXiv:2402.01799*, 2024.

T. Chen, Y. Sui, X. Chen, A. Zhang, and Z. Wang. A unified lottery ticket hypothesis for graph neural networks. In *International Conference on Machine Learning*, pages 1695–1706. PMLR, 2021.

Y. Chen, T. Luo, S. Liu, S. Zhang, L. He, J. Wang, L. Li, T. Chen, Z. Xu, N. Sun, et al. Dadiannao: A machine-learning supercomputer. In *2014 47th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 609–622. IEEE, 2014.

Y. Chen, Y. Xie, L. Song, F. Chen, and T. Tang. A Survey of Accelerator Architectures for Deep Neural Networks. *Engineering*, 6(3):264–274, 2020. ISSN 20958099. . URL https://doi.org/10.1016/j.eng.2020.01.007.

Y.-H. Chen, T. Krishna, J. S. Emer, and V. Sze. Eyeriss: An energy-efficient reconfigurable accelerator for deep convolutional neural networks. *IEEE journal of solid-state circuits*, 52(1):127–138, 2016.

Y. H. Chen, T. Krishna, J. S. Emer, and V. Sze. Eyeriss: An Energy-Efficient Reconfigurable Accelerator for Deep Convolutional Neural Networks. *IEEE J. Solid-State Circuits*, 52(1):127–138, jan 2017. ISSN 00189200. .

Y.-H. Chen, T.-J. Yang, J. Emer, and V. Sze. Eyeriss v2: A flexible accelerator for emerging deep neural networks on mobile devices. *IEEE Journal on Emerging and Selected Topics in Circuits and Systems*, 9(2):292–308, 2019.

F. Chollet. Xception: Deep learning with depthwise separable convolutions. In *Proc. - 30th IEEE Conf. Comput. Vis. Pattern Recognition, CVPR 2017*, 2017a. ISBN 9781538604571. .

F. Chollet. Xception: Deep learning with depthwise separable convolutions. In *Proc. - 30th IEEE Conf. Comput. Vis. Pattern Recognition, CVPR 2017*, 2017b. ISBN 9781538604571. .

M. Courbariaux, Y. Bengio, and J. P. David. Binaryconnect: Training deep neural networks with binary weights during propagations. In *Adv. Neural Inf. Process. Syst.*, 2015.

D. Das, N. Mellempudi, D. Mudigere, D. Kalamkar, S. Avancha, K. Banerjee, S. Sridharan, K. Vaidyanathan, B. Kaul, E. Georganas, et al. Mixed precision training of convolutional neural networks using integer operations. *arXiv preprint arXiv:1802.00930*, 2018.

E. Daxberger, E. Nalisnick, J. U. Allingham, J. Antorán, and J. M. Hernández-Lobato. Bayesian deep learning via subnetwork inference. In *International Conference on Machine Learning*, pages 2510–2521. PMLR, 2021.

E. L. Denton, W. Zaremba, J. Bruna, Y. LeCun, and R. Fergus. Exploiting linear structure within convolutional networks for efficient evaluation. In *Advances in neural information processing systems*, pages 1269–1277, 2014.

R. Dorrance, F. Ren, and D. Marković. A scalable sparse matrix-vector multiplication kernel for energy-efficient sparse-blas on FPGAs. In *ACM/SIGDA Int. Symp. F. Program. Gate Arrays - FPGA*, pages 161–169. Association for Computing Machinery, 2014. ISBN 9781450326711. .

Z. Du, R. Fasthuber, T. Chen, P. Ienne, L. Li, T. Luo, X. Feng, Y. Chen, and O. Temam. Shidiannao: Shifting vision processing closer to the sensor. In *Proceedings of the 42nd Annual International Symposium on Computer Architecture*, pages 92–104, 2015.

T. Elsken, J. H. Metzen, and F. Hutter. Neural architecture search: A survey. *J. Mach. Learn. Res.*, 2019. ISSN 15337928.

S. K. Esser, J. L. McKinstry, D. Bablani, R. Appuswamy, and D. S. Modha. Learned Step Size Quantization. *8th Int. Conf. Learn. Represent. ICLR 2020*, 2020. URL http://arxiv.org/abs/1902.08153.

A. Fan, P. Stock, B. Graham, E. Grave, R. Gribonval, H. Jegou, and A. Joulin. Training with Quantization Noise for Extreme Model Compression. *9th Int. Conf. Learn. Represent. ICLR 2021 - Conf. Track Proc.*, apr 2021. URL http://arxiv.org/abs/2004.07320.

V. Feldman and C. Zhang. What neural networks memorize and why: Discovering the long tail via influence estimation. *Advances in Neural Information Processing Systems*, 33:2881–2891, 2020.

Fireworks AI. Fireworks ai: Build and deploy large ai models, 2024. URL https://fireworks.ai/.

J. Frankle and M. Carbin. The lottery ticket hypothesis: Finding sparse, trainable neural networks. In *7th Int. Conf. Learn. Represent. ICLR 2019*, 2019.

J. Frankle, G. K. Dziugaite, D. M. Roy, and M. Carbin. Stabilizing the Lottery Ticket Hypothesis. 2019. URL http://arxiv.org/abs/1903.01611.

E. Frantar, S. Ashkboos, T. Hoefler, and D. Alistarh. Gptq: Accurate post-training quantization for generative pre-trained transformers. *arXiv preprint arXiv:2210.17323*, 2022.

K. Fukushima and S. Miyake. Neocognitron: A Self-Organizing Neural Network Model for a Mechanism of Visual Pattern Recognition. pages 267–285. Springer, Berlin, Heidelberg, 1982. . URL https://link.springer.com/chapter/10.1007/978-3-642-46466-9{_}18.

Y. Gal and Z. Ghahramani. Dropout as a bayesian approximation: Representing model uncertainty in deep learning. In *international conference on machine learning*, pages 1050–1059. PMLR, 2016.

Y. Gal, R. Islam, and Z. Ghahramani. Deep bayesian active learning with image data. In *International conference on machine learning*, pages 1183–1192. PMLR, 2017.

Y. Gal et al. Uncertainty in deep learning. 2016.

T. Gale, M. Zaharia, C. Young, and E. Elsen. Sparse gpu kernels for deep learning. In *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–14. IEEE, 2020.

R. Gallager. Variations on a theme by huffman. *IEEE Transactions on Information Theory*, 24(6):668–674, 1978.

J. Gawlikowski, C. R. N. Tassi, M. Ali, J. Lee, M. Humt, J. Feng, A. Kruspe, R. Triebel, P. Jung, R. Roscher, et al. A survey of uncertainty in deep neural networks. *arXiv preprint arXiv:2107.03342*, 2021.

S. Girish, S. R. Maiya, K. Gupta, H. Chen, L. S. Davis, and A. Shrivastava. The lottery ticket hypothesis for object recognition. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 762–771, 2021.

J. Gou, B. Yu, S. J. Maybank, and D. Tao. Knowledge distillation: A survey. *International Journal of Computer Vision*, 129(6):1789–1819, 2021.

A. Graves. Practical variational inference for neural networks. *Advances in neural information processing systems*, 24, 2011.

Groq. What is an lpu? Whitepaper, Groq, 7 2024. URL https://wow.groq.com/wp-content/uploads/2024/07/GroqThoughts_WhatIsALPU-vF.pdf.

C. Guo, G. Pleiss, Y. Sun, and K. Q. Weinberger. On calibration of modern neural networks. In *International Conference on Machine Learning*, pages 1321–1330. PMLR, 2017.

R. Hameed, W. Qadeer, M. Wachs, O. Azizi, A. Solomatnikov, B. C. Lee, S. Richardson, C. Kozyrakis, and M. Horowitz. Understanding sources of inefficiency in general-purpose chips. In *Proceedings of the 37th annual international symposium on Computer architecture*, pages 37–47, 2010.

S. Han, J. Pool, J. Tran, and W. Dally. Learning both weights and connections for efficient neural network. In *Advances in neural information processing systems*, pages 1135–1143, 2015a.

S. Han, J. Pool, J. Tran, and W. J. Dally. Learning both weights and connections for efficient neural networks. In *Adv. Neural Inf. Process. Syst.*, 2015b.

S. Han, X. Liu, H. Mao, J. Pu, A. Pedram, M. A. Horowitz, and W. J. Dally. EIE: Efficient Inference Engine on Compressed Deep Neural Network. feb 2016a. URL http://arxiv.org/abs/1602.01528.

S. Han, H. Mao, and W. J. Dally. Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding. *arXiv preprint arXiv:1510.00149*, 2016b.

R. Hashemian. Design and hardware implementation of a memory efficient huffman decoding. *IEEE Transactions on Consumer Electronics*, 40(3):345–352, 1994.

K. He, X. Zhang, S. Ren, and J. Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 770–778, 2016.

Y. He, J. Lin, Z. Liu, H. Wang, L.-J. Li, and S. Han. Amc: Automl for model compression and acceleration on mobile devices. In *Proceedings of the European Conference on Computer Vision (ECCV)*, pages 784–800, 2018.

S. Hecker, D. Dai, and L. Van Gool. Failure prediction for autonomous driving. In *2018 IEEE Intelligent Vehicles Symposium (IV)*, pages 1792–1799. IEEE, 2018.

D. Hendrycks and K. Gimpel. A baseline for detecting misclassified and out-of-distribution examples in neural networks. *arXiv preprint arXiv:1610.02136*, 2016.

D. Hendrycks, K. Lee, and M. Mazeika. Using pre-training can improve model robustness and uncertainty. In *International Conference on Machine Learning*, pages 2712–2721. PMLR, 2019.

D. Hendrycks, S. Basart, M. Mazeika, M. Mostajabi, J. Steinhardt, and D. Song. Scaling out-of-distribution detection for real-world settings. *arXiv preprint arXiv:2104.14947*, 2021.

P. Hennessy. A new golden age for computer architecture: Domain-specific hardware/software co-design, enhanced security, open instruction sets, and agile chip development. In *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*, pages 27–29, 2018. .

G. Hinton, O. Vinyals, and J. Dean. Distilling the Knowledge in a Neural Network. pages 1–9, 2015. URL http://arxiv.org/abs/1503.02531.

M. Horowitz. 1.1 Computing's energy problem (and what we can do about it). In *Dig. Tech. Pap. - IEEE Int. Solid-State Circuits Conf.*, 2014. ISBN 9781479909186. .

A. G. Howard, M. Zhu, B. Chen, D. Kalenichenko, W. Wang, T. Weyand, M. Andreetto, and H. Adam. MobileNet V1. *arXiv Prepr. arXiv1704.04861*, 2017. ISSN 0004-6361. .

C. Huang, P. Liu, and L. Fang. Mxqn: Mixed quantization for reducing bit-width of weights and activations in deep convolutional neural networks. *Applied Intelligence*, 51(7):4561–4574, 2021.

G. Huang, Z. Liu, L. Van Der Maaten, and K. Q. Weinberger. Densely connected convolutional networks. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 4700–4708, 2017.

I. Hubara, M. Courbariaux, D. Soudry, R. El-Yaniv, and Y. Bengio. Binarized neural networks. In *Adv. Neural Inf. Process. Syst.*, 2016.

D. H. Hubel and T. N. Wiesel. Receptive fields, binocular interaction and functional architecture in the cat's visual cortex. *J. Physiol.*, 160(1):106–154, jan 1962. ISSN 14697793. .

D. A. Huffman. A method for the construction of minimum-redundancy codes. *Proceedings of the IRE*, 40(9):1098–1101, 1952.

P. R. Huttenlocher et al. Synaptic density in human frontal cortex-developmental changes and effects of aging. *Brain Res*, 163(2):195–205, 1979.

F. N. Iandola, M. W. Moskewicz, K. Ashraf, S. Han, W. J. Dally, and K. Keutzer. SqueezeNet. *arXiv*, 2016.

N. N. C. S. IEEE and S. S. IEEE. IEEE-INNS International Joint Conference on Neural Networks, Baltimore, 1992. page 4012, 1992.

P. Izmailov, D. Podoprikhin, T. Garipov, D. Vetrov, and A. G. Wilson. Averaging weights leads to wider optima and better generalization. *Uncertainty in Artificial Intelligence*, pages 876–885, 2018.

B. Jacob, S. Kligys, B. Chen, M. Zhu, M. Tang, A. Howard, H. Adam, and D. Kalenichenko. Quantization and Training of Neural Networks for Efficient Integer-Arithmetic-Only Inference. In *Proc. IEEE Comput. Soc. Conf. Comput. Vis. Pattern Recognit.*, pages 2704–2713, 2018. ISBN 9781538664209. .

Q. Jin, L. Yang, and Z. Liao. Adabits: Neural network quantization with adaptive bit-widths. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 2146–2156, 2020.

M. I. Jordan and T. M. Mitchell. Machine learning: Trends, perspectives, and prospects. *Science*, 349(6245):255–260, 2015.

N. P. Jouppi, C. Young, N. Patil, D. Patterson, G. Agrawal, R. Bajwa, S. Bates, S. Bhatia, N. Boden, A. Borchers, R. Boyle, P. L. Cantin, C. Chao, C. Clark, J. Coriell, M. Daley, M. Dau, J. Dean, B. Gelb, T. V. Ghaemmaghami, R. Gottipati, W. Gulland, R. Hagmann, C. Richard Ho, D. Hogberg, J. Hu, R. Hundt, D. Hurt, J. Ibarz, A. Jaffey, A. Jaworski, A. Kaplan, H. Khaitan, D. Killebrew, A. Koch, N. Kumar, S. Lacy, J. Laudon, J. Law, D. Le, C. Leary, Z. Liu, K. Lucke, A. Lundin, G. MacKean, A. Maggiore, M. Mahony, K. Miller, R. Nagarajan, R. Narayanaswami, R. Ni, K. Nix, T. Norrie, M. Omernick, N. Penukonda, A. Phelps, J. Ross, M. Ross, A. Salek, E. Samadiani, C. Severn, G. Sizikov, M. Snelham, J. Souter, D. Steinberg, A. Swing, M. Tan, G. Thorson, B. Tian, H. Toma, E. Tuttle, V. Vasudevan, R. Walter, W. Wang, E. Wilcox, and D. H. Yoon. In-datacenter performance analysis of a tensor processing unit. In *Proc. - Int. Symp. Comput. Archit.*, 2017a. ISBN 9781450348928. .

N. P. Jouppi, C. Young, N. Patil, D. Patterson, G. Agrawal, R. Bajwa, S. Bates, S. Bhatia, N. Boden, A. Borchers, et al. In-datacenter performance analysis of a tensor processing unit. In *Proceedings of the 44th annual international symposium on computer architecture*, pages 1–12, 2017b.

N. P. Jouppi, D. Hyun Yoon, M. Ashcraft, M. Gottscho, T. B. Jablin, G. Kurian, J. Laudon, S. Li, P. Ma, X. Ma, T. Norrie, N. Patil, S. Prasad, C. Young, Z. Zhou, and D. Patterson. Ten Lessons From Three Generations Shaped Google's TPUv4i : Industrial Product. *2021 ACM/IEEE 48th Annu. Int. Symp. Comput. Archit.*, pages 1–14, jun 2021. . URL https://ieeexplore.ieee.org/document/9499913/.

S. Jung, C. Son, S. Lee, J. Son, J. J. Han, Y. Kwak, S. J. Hwang, and C. Choi. Learning to quantize deep networks by optimizing quantization intervals with

task loss. In *Proc. IEEE Comput. Soc. Conf. Comput. Vis. Pattern Recognit.*, 2019a. ISBN 9781728132938. .

S. Jung, C. Son, S. Lee, J. Son, J.-J. Han, Y. Kwak, S. J. Hwang, and C. Choi. Learning to quantize deep networks by optimizing quantization intervals with task loss. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 4350–4359, 2019b.

A. Jungo, R. Meier, E. Ermis, E. Herrmann, and M. Reyes. Uncertainty-driven sanity check: application to postoperative brain tumor cavity segmentation. *arXiv preprint arXiv:1806.03106*, 2018.

J. Kaplan, S. McCandlish, T. Henighan, T. B. Brown, B. Chess, R. Child, S. Gray, A. Radford, J. Wu, and D. Amodei. Scaling laws for neural language models. *arXiv preprint arXiv:2001.08361*, 2020.

S. W. Keckler, W. J. Dally, B. Khailany, M. Garland, and D. Glasco. Gpus and the future of parallel computing. *IEEE micro*, 31(5):7–17, 2011.

A. Kendall and Y. Gal. What uncertainties do we need in bayesian deep learning for computer vision? *Advances in Neural Information Processing Systems*, 30, 2017.

G. Kestor, R. Gioiosa, D. J. Kerbyson, and A. Hoisie. Quantifying the energy cost of data movement in scientific applications. In *2013 IEEE international symposium on workload characterization (IISWC)*, pages 56–65. IEEE, 2013.

D. P. Kingma and J. L. Ba. Adam: A method for stochastic optimization. In *3rd Int. Conf. Learn. Represent. ICLR 2015 - Conf. Track Proc.*, 2015.

A. Kristiadi, M. Hein, and P. Hennig. Learnable uncertainty under laplace approximations. In *Uncertainty in Artificial Intelligence*, pages 344–353. PMLR, 2021.

A. Krizhevsky, I. Sutskever, and G. E. Hinton. 2012 AlexNet. *Adv. Neural Inf. Process. Syst.*, 2012. ISSN 10495258. .

H. Kwon, A. Samajdar, and T. Krishna. Maeri: Enabling flexible dataflow mapping over dnn accelerators via reconfigurable interconnects. *ACM SIGPLAN Notices*, 53(2):461–475, 2018.

B. Lakshminarayanan, A. Pritzel, and C. Blundell. Simple and scalable predictive uncertainty estimation using deep ensembles. *Advances in neural information processing systems*, 30, 2017.

S. Laughlin. A simple coding procedure enhances a neuron's information capacity. *Zeitschrift für Naturforschung c*, 36(9-10):910–912, 1981.

Y. LeCun, B. Boser, J. S. Denker, D. Henderson, R. E. Howard, W. Hubbard, and L. D. Jackel. Backpropagation applied to handwritten zip code recognition. *Neural computation*, 1(4):541–551, 1989.

Y. LeCun, J. S. Denker, and S. A. Solla. Optimal brain damage. In *Adv. Neural Inf. Process. Syst.*, pages 598–605, 1990.

J. Lee, C. Kim, S. Kang, D. Shin, S. Kim, and H.-J. Yoo. Unpu: An energy-efficient deep neural network accelerator with fully variable weight bit precision. *IEEE Journal of Solid-State Circuits*, 54(1):173–185, 2018.

J. Lee, D. Kim, and B. Ham. Network Quantization with Element-wise Gradient Scaling. 2021. URL http://arxiv.org/abs/2104.00903.

N. Lee, T. Ajanthan, and P. H. Torr. SnIP: Single-shot network pruning based on connection sensitivity. In *7th Int. Conf. Learn. Represent. ICLR 2019*, 2019.

H. Li, B. Gao, Z. Chen, Y. Zhao, P. Huang, H. Ye, L. Liu, X. Liu, and J. Kang. A learnable parallel processing architecture towards unity of memory and computing. *Sci. Rep.*, 2015. ISSN 20452322. .

H. Li, A. Kadav, I. Durdanovic, H. Samet, and H. P. Graf. Pruning filters for efficient convnets. *arXiv preprint arXiv:1608.08710*, 2016.

Y. Li, S. Xu, B. Zhang, X. Cao, P. Gao, and G. Guo. Q-vit: Accurate and fully quantized low-bit vision transformer. *Advances in Neural Information Processing Systems*, 35:34451–34463, 2022.

J. Lin, Y. Rao, J. Lu, and J. Zhou. Runtime neural pruning. In *Adv. Neural Inf. Process. Syst.*, 2017.

J. Lin, W.-M. Chen, Y. Lin, C. Gan, S. Han, et al. Mcunet: Tiny deep learning on iot devices. *Advances in Neural Information Processing Systems*, 33:11711–11722, 2020a.

J. Lin, J. Tang, H. Tang, S. Yang, W.-M. Chen, W.-C. Wang, G. Xiao, X. Dang, C. Gan, and S. Han. Awq: Activation-aware weight quantization for on-device llm compression and acceleration. *Proceedings of Machine Learning and Systems*, 6: 87–100, 2024.

T. Lin, L. Barba, and M. Jaggi. Dynamic Model Pruning with Fre. *ICLR*, 2020b.

M. Liu, T.-D. Ene, R. Kirby, C. Cheng, N. Pinckney, R. Liang, J. Alben, H. Anand, S. Banerjee, I. Bayraktaroglu, et al. Chipnemo: Domain-adapted llms for chip design. *arXiv preprint arXiv:2311.00176*, 2023.

Z. Liu, J. Li, Z. Shen, G. Huang, S. Yan, and C. Zhang. Learning Efficient Convolutional Networks through Network Slimming. In *Proc. IEEE Int. Conf. Comput. Vis.*, 2017. ISBN 9781538610329. .

E. S. Lubana and R. P. Dick. A Gradient Flow Framework For Analyzing Network Pruning. pages 1–18, 2020. URL http://arxiv.org/abs/2009.11839.

S. Ma, H. Wang, L. Ma, L. Wang, W. Wang, S. Huang, L. Dong, R. Wang, J. Xue, and F. Wei. The era of 1-bit llms: All large language models are in 1.58 bits. *arXiv preprint arXiv:2402.17764*, 2024.

D. J. C. Mackay. *Bayesian methods for adaptive models*. California Institute of Technology, 1992.

W. J. Maddox, P. Izmailov, T. Garipov, D. P. Vetrov, and A. G. Wilson. A simple baseline for bayesian uncertainty in deep learning. *Advances in neural information processing systems*, 32, 2019.

H. Mao and W. J. Dally. Deep Compression: Compressing Deep Neural. *4th Int. Conf. Learn. Represent. ICLR*, 2016a.

H. Mao and W. J. Dally. Deep Compression: Compressing Deep Neural. *Iclr 2016*, pages 1–14, 2016b. URL https://arxiv.org/abs/1510.00149.

R. McAllister, Y. Gal, A. Kendall, M. Van Der Wilk, A. Shah, R. Cipolla, and A. Weller. Concrete problems for autonomous vehicle safety: Advantages of bayesian deep learning. In *Proceedings of the Twenty-Sixth International Joint Conference on Artificial Intelligence*. International Joint Conferences on Artificial Intelligence Organization, 2017.

L. McInnes, J. Healy, and J. Melville. Umap: Uniform manifold approximation and projection for dimension reduction. *arXiv preprint arXiv:1802.03426*, 2018.

S. A. McKee and R. W. Wisniewski. *Memory Wall*, pages 1110–1116. Springer US, Boston, MA, 2011. ISBN 978-0-387-09766-4. . URL https://doi.org/10.1007/978-0-387-09766-4_234.

Meta AI. Llama 3.1: Open foundation and fine-tuned chat models, 2024. URL https://ai.meta.com/blog/meta-llama-3-1/.

L. Mi, H. Wang, Y. Tian, and N. Shavit. Training-free uncertainty estimation for neural networks. 2019.

A. Moffat. Huffman coding. *ACM Computing Surveys (CSUR)*, 52(4):1–35, 2019.

P. Molchanov, S. Tyree, T. Karras, T. Aila, and J. Kautz. Pruning Convolutional Neural Networks for Resource Efficient Inference. nov 2016. URL http://arxiv.org/abs/1611.06440.

P. Molchanov, A. Mallya, S. Tyree, I. Frosio, and J. Kautz. Importance Estimation for Neural Network Pruning. jun 2019a. URL http://arxiv.org/abs/1906.10771.

P. Molchanov, A. Mallya, S. Tyree, I. Frosio, and J. Kautz. Importance estimation for neural network pruning. In *Proc. IEEE Conf. Comput. Vis. Pattern Recognit.*, pages 11264–11272, 2019b.

B. Moons and M. Verhelst. A 0.3-2.6 TOPS/W precision-scalable processor for real-time large-scale ConvNets. *IEEE Symp. VLSI Circuits, Dig. Tech. Pap.*, 2016-Septe:1–2, 2016a. .

B. Moons and M. Verhelst. A 0.3–2.6 tops/w precision-scalable processor for real-time large-scale convnets. In *2016 IEEE Symposium on VLSI Circuits (VLSI-Circuits)*, pages 1–2. IEEE, 2016b.

G. E. Moore. Cramming more components onto integrated circuits. *Proceedings of the IEEE*, 86(1):82–85, 1998.

A. S. Morcos, H. Yu, M. Paganini, and Y. Tian. One ticket to win them all: generalizing lottery ticket initializations across datasets and optimizers. (NeurIPS), 2019. URL http://arxiv.org/abs/1906.02773.

K. Muhammad, A. Ullah, J. Lloret, J. Del Ser, and V. H. C. de Albuquerque. Deep learning for safe autonomous driving: Current challenges and future directions. *IEEE Transactions on Intelligent Transportation Systems*, 22(7):4316–4336, 2020.

M. P. Naeini, G. Cooper, and M. Hauskrecht. Obtaining well calibrated probabilities using bayesian binning. In *Proceedings of the AAAI conference on artificial intelligence*, volume 29, 2015.

M. Naumov, L. Chien, P. Vandermersch, and U. Kapasi. Cusparse library. In *GPU Technology Conference*, 2010.

R. M. Neal. *Bayesian learning for neural networks*, volume 118. Springer Science & Business Media, 2012.

A. Niculescu-Mizil and R. Caruana. Predicting good probabilities with supervised learning. In *Proceedings of the 22nd international conference on Machine learning*, pages 625–632, 2005.

NVIDIA. NVDLA Primer — NVDLA Documentation, 2018. URL http://nvdla.org/primer.htmlhttp://nvdla.org/primer.html{%}0Ahttp://nvdla.org/hw/v1/ias/unit{_}description.html.

S. Oh, H. Sim, S. Lee, and J. Lee. Automated Log-Scale Quantization for Low-Cost Deep Neural Networks. *Proc. IEEE/CVF Conf. Comput. Vis. Pattern Recognit.*, 2021a.

S. Oh, H. Sim, S. Lee, and J. Lee. Automated log-scale quantization for low-cost deep neural networks. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 742–751, 2021b.

I. Osband, C. Blundell, A. Pritzel, and B. Van Roy. Deep exploration via bootstrapped dqn. *Advances in neural information processing systems*, 29, 2016.

Y. Ovadia, E. Fertig, J. Ren, Z. Nado, D. Sculley, S. Nowozin, J. Dillon, B. Lakshminarayanan, and J. Snoek. Can you trust your model's uncertainty? evaluating predictive uncertainty under dataset shift. *Advances in Neural Information Processing Systems*, 32, 2019.

D. Pandiyan and C.-J. Wu. Quantifying the energy cost of data movement for emerging smart phone workloads on mobile platforms. In *2014 IEEE International Symposium on Workload Characterization (IISWC)*, pages 171–180. IEEE, 2014.

A. Parashar, M. Rhu, A. Mukkara, A. Puglielli, R. Venkatesan, B. Khailany, J. Emer, S. W. Keckler, and W. J. Dally. Scnn: An accelerator for compressed-sparse convolutional neural networks. *ACM SIGARCH computer architecture news*, 45 (2):27–40, 2017.

T. Pearce, A. Brintrup, M. Zaki, and A. Neely. High-quality prediction intervals for deep learning: A distribution-free, ensembled approach. In *International conference on machine learning*, pages 4075–4084. PMLR, 2018.

H. Pham, M. Guan, B. Zoph, Q. Le, and J. Dean. Efficient neural architecture search via parameters sharing. In *International conference on machine learning*, pages 4095–4104. PMLR, 2018.

M. A. Pimentel, D. A. Clifton, L. Clifton, and L. Tarassenko. A review of novelty detection. *Signal Processing*, 99:215–249, 2014.

J. Platt et al. Probabilistic outputs for support vector machines and comparisons to regularized likelihood methods. *Advances in large margin classifiers*, 10(3):61–74, 1999.

M. Rastegari, V. Ordonez, J. Redmon, and A. Farhadi. XNOR-net: Imagenet classification using binary convolutional neural networks. *Lect. Notes Comput. Sci. (including Subser. Lect. Notes Artif. Intell. Lect. Notes Bioinformatics)*, 9908 LNCS:525–542, 2016. ISSN 16113349. .

B. Reagen, P. Whatmough, R. Adolf, S. Rama, H. Lee, S. K. Lee, J. Miguel Hernández-Lobato, G.-Y. Wei, and D. Brooks. Minerva: Enabling Low-Power, Highly-Accurate Deep Neural Network Accelerators. Technical report.

M. T. Ribeiro, S. Singh, and C. Guestrin. " why should i trust you?" explaining the predictions of any classifier. In *Proceedings of the 22nd ACM SIGKDD international conference on knowledge discovery and data mining*, pages 1135–1144, 2016.

A.-J. Rousseau, T. Becker, J. Bertels, M. B. Blaschko, and D. Valkenborg. Post training uncertainty calibration of deep networks for medical image segmentation. In *2021 IEEE 18th International Symposium on Biomedical Imaging (ISBI)*, pages 1052–1056. IEEE, 2021.

N. Roy and A. McCallum. Toward optimal active learning through monte carlo estimation of error reduction. *ICML, Williamstown*, 2:441–448, 2001.

L. Ruff, J. R. Kauffmann, R. A. Vandermeulen, G. Montavon, W. Samek, M. Kloft, T. G. Dietterich, and K.-R. Müller. Unifying review of deep and shallow anomaly detection. *Proceedings of the IEEE*, 109(5):756–795, 2021.

D. E. Rumelhart, G. E. Hinton, and R. J. Williams. Learning representations by back-propagating errors. *nature*, 323(6088):533–536, 1986.

E. Sackinger, B. E. Boser, J. Bromley, Y. LeCun, and L. D. Jackel. Application of the ANNA Neural Network Chip to High-Speed Character Recognition, 1992. ISSN 19410093.

N. Sato and W. Tinney. Techniques for exploiting the sparsity or the network admittance matrix. *IEEE Transactions on Power Apparatus and Systems*, 82(69): 944–950, 1963.

A. Sebastian, M. Le Gallo, R. Khaddam-Aljameh, and E. Eleftheriou. Memory devices and applications for in-memory computing, 2020. ISSN 17483395.

B. Settles. From theories to queries: Active learning in practice. In *Active learning and experimental design workshop in conjunction with AISTATS 2010*, pages 1–18. JMLR Workshop and Conference Proceedings, 2011.

C. E. Shannon. A Mathematical Theory of Communication. *Bell Syst. Tech. J.*, 27(3): 379–423, 1948a. ISSN 15387305. .

C. E. Shannon. A mathematical theory of communication. *The Bell system technical journal*, 27(3):379–423, 1948b.

H. Sharma, J. Park, N. Suda, L. Lai, B. Chau, J. K. Kim, V. Chandra, and H. Esmaeilzadeh. Bit fusion: Bit-level dynamically composable architecture for accelerating deep neural network. In *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*, pages 764–775. IEEE, 2018.

M. Sharma, S. Farquhar, E. Nalisnick, and T. Rainforth. Do bayesian neural networks need to be fully stochastic? In *International Conference on Artificial Intelligence and Statistics*, pages 7694–7722. PMLR, 2023.

L. Smith and Y. Gal. Understanding measures of uncertainty for adversarial example detection. *arXiv preprint arXiv:1803.08533*, 2018.

G. Stix. Encoding theneatness' of ones and zeroes. *Scientific American*, 265(3): 54–55, 1991.

P. Stock, A. Joulin, R. Gribonval, B. Graham, and H. Jégou. And the bit goes down: Revisiting the quantization of neural networks, 2020. ISSN 23318422.

C. Subia-Waud and S. Dasmahapatra. Weight fixing networks. In *European Conference on Computer Vision*, pages 415–431. Springer, 2022.

V. Sze, Y.-H. Chen, T.-J. Yang, and J. Emer. Efficient Processing of Deep Neural Networks: A Tutorial and Survey. mar 2017. URL http://arxiv.org/abs/1703.09039.

V. Sze, Y.-H. Chen, T.-J. Yang, and J. S. Emer. Efficient Processing of Deep Neural Networks. *Synth. Lect. Comput. Archit.*, 2020. ISSN 1935-3235. .

C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, and A. Rabinovich. GoogLeNet. *Proc. IEEE Comput. Soc. Conf. Comput. Vis. Pattern Recognit.*, 2014. ISSN 10636919. .

N. Tagasovska and D. Lopez-Paz. Single-model uncertainties for deep learning. *Advances in Neural Information Processing Systems*, 32, 2019.

M. Tan and Q. V. Le. EfficientNet: Rethinking Model Scaling for Convolutional Neural Networks. may 2019. URL http://arxiv.org/abs/1905.11946.

M. Tan, B. Chen, R. Pang, V. Vasudevan, M. Sandler, A. Howard, and Q. V. Le. Mnasnet: Platform-aware neural architecture search for mobile. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 2820–2828, 2019.

E. Tartaglione, S. Lathuilière, A. Fiandrotti, M. Cagnazzo, and M. Grangetto. HEMP: High-order Entropy Minimization for neural network comPression. 2021. . URL http://arxiv.org/abs/2107.05298{%}0Ahttp://dx.doi.org/10.1016/j.neucom.2021.07.022.

G. Z. Tau and B. S. Peterson. Normal development of brain circuits. *Neuropsychopharmacology*, 35(1):147–168, 2010.

J. Tian, C. Rivera, S. Di, J. Chen, X. Liang, D. Tao, and F. Cappello. Revisiting huffman coding: Toward extreme performance on modern gpu architectures. *IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 1–11, 2021.

H. Touvron, M. Cord, M. Douze, F. Massa, A. Sablayrolles, and H. Jégou. Training data-efficient image transformers & distillation through attention. In *International conference on machine learning*, pages 10347–10357. PMLR, 2021.

UnslothAI. Unsloth: A pytorch library for implementing various machine learning models and algorithms. https://github.com/unslothai/unsloth, 2024. Accessed: 2024-10-05.

J. Vaicenavicius, D. Widmann, C. Andersson, F. Lindsten, J. Roll, and T. Schön. Evaluating model calibration in classification. In *The 22nd International Conference on Artificial Intelligence and Statistics*, pages 3459–3467. PMLR, 2019.

L. Van der Maaten and G. Hinton. Visualizing data using t-sne. *Journal of machine learning research*, 9(11), 2008.

K. Wang, Z. Liu, Y. Lin, J. Lin, and S. Han. Haq: Hardware-aware automated quantization with mixed precision. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 8612–8620, 2019.

T. A. Welch. A technique for high-performance data compression. *Computer*, 17 (06):8–19, 1984.

M. Welling and Y. W. Teh. Bayesian learning via stochastic gradient langevin dynamics. In *Proceedings of the 28th international conference on machine learning (ICML-11)*, pages 681–688, 2011.

S. Wold, K. Esbensen, and P. Geladi. Principal component analysis. *Chemometrics and intelligent laboratory systems*, 2(1-3):37–52, 1987.

J. Wu, Y. Wang, Z. Wu, Z. Wang, A. Veeraraghavan, and Y. Lin. Deep κ-means: Re-training and parameter sharing with harder cluster assignments for compressing deep convolutions. *35th Int. Conf. Mach. Learn. ICML 2018*, 12: 8523–8532, 2018.

K. Yamamoto. Learnable Companding Quantization for Accurate Low-bit Neural Networks. pages 5029–5038, 2021. URL http://arxiv.org/abs/2103.07156.

J. Yang, K. Zhou, Y. Li, and Z. Liu. Generalized out-of-distribution detection: A survey. *arXiv preprint arXiv:2110.11334*, 2021.

T. J. Yang, Y. H. Chen, and V. Sze. Designing energy-efficient convolutional neural networks using energy-aware pruning. In *Proc. - 30th IEEE Conf. Comput. Vis. Pattern Recognition, CVPR 2017*, 2017a. ISBN 9781538604571. .

T.-J. Yang, Y.-H. Chen, and V. Sze. Designing energy-efficient convolutional neural networks using energy-aware pruning. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 5687–5695, 2017b.

H. Yoshizawa and D. Takahashi. Automatic tuning of sparse matrix-vector multiplication for CRS format on GPUs. In *Proc. - 15th IEEE Int. Conf. Comput. Sci. Eng. CSE 2012 10th IEEE/IFIP Int. Conf. Embed. Ubiquitous Comput. EUC 2012*, 2012. ISBN 9780769549149. .

H. Yu, S. Edunov, Y. Tian, and A. S. Morcos. Playing the lottery with rewards and multiple languages: lottery tickets in RL and NLP. 2019. URL http://arxiv.org/abs/1906.02768.

X. Yu, T. Liu, X. Wang, and D. Tao. On compressing deep models by low rank and sparse decomposition. *Proc. - 30th IEEE Conf. Comput. Vis. Pattern Recognition, CVPR 2017*, 2017-Janua:67–76, 2017. .

Z. Yu, Y. Shi, T. Huang, and Y. Yu. Kernel Quantization for Efficient Network Compression. 2020. URL http://arxiv.org/abs/2003.05148.

W. W. Yuhang Li, Xin Dong. Additive Powers-of-Two Quantization: an Efficient Non-Uniform Discretization for Neural Networks. *Iclr*, (2014):2016–2019, 2020. URL https://openreview.net/pdf?id=BkgXT24tDS.

D. Zhang, J. Yang, D. Ye, and G. Hua. LQ-Nets: Learned quantization for highly accurate and compact deep neural networks. In *Lect. Notes Comput. Sci. (including Subser. Lect. Notes Artif. Intell. Lect. Notes Bioinformatics)*, volume 11212 LNCS, pages 373–390, 2018. ISBN 9783030012366. . URL https://github.com/Microsoft/LQ-Nets.

S. Zhang, Z. Du, L. Zhang, H. Lan, S. Liu, L. Li, Q. Guo, T. Chen, and Y. Chen. Cambricon-x: An accelerator for sparse neural networks. In *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 1–12. IEEE, 2016.

S. Zheng, Y. Shen, C. Zhu, J. Wen, Q. Yu, and X. Liu. Out-of-distribution detection for reliable reinforcement learning. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 2020.

A. Zhou, A. Yao, Y. Guo, L. Xu, and Y. Chen. Incremental network quantization: Towards lossless cnns with low-precision weights, 2017.

H. Zhou, J. Lan, R. Liu, and J. Yosinski. Deconstructing Lottery Tickets: Zeros, Signs, and the Supermask. (NeurIPS), 2019. URL http://arxiv.org/abs/1905.01067.

S. Zhou, Y. Wu, Z. Ni, X. Zhou, H. Wen, and Y. Zou. DoReFa-Net: Training Low Bitwidth Convolutional Neural Networks with Low Bitwidth Gradients. jun 2016a. URL http://arxiv.org/abs/1606.06160.

S. Zhou, Y. Wu, Z. Ni, X. Zhou, H. Wen, and Y. Zou. Dorefa-net: Training low bitwidth convolutional neural networks with low bitwidth gradients. *arXiv preprint arXiv:1606.06160*, 2016b.

B. Zoph and Q. V. Le. Neural architecture search with reinforcement learning. *arXiv preprint arXiv:1611.01578*, 2016.