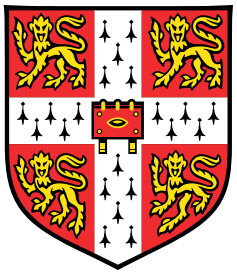
Neural Network Compression



Osman Khawer Zubair

Department of Engineering
University of Cambridge

This dissertation is submitted for the degree of
Master of Philosophy in Machine Learning, Speech and Language Technology

Darwin College August 2018

I would like to dedicate this to my mother, grandfather and late grandmother who have always supported me.

Declaration

I hereby declare that except where specific reference is made to the work of others, the contents of this dissertation are original and have not been submitted in whole or in part for consideration for any other degree or qualification in this, or any other university. This dissertation is my own work and contains nothing which is the outcome of work done in collaboration with others, except as specified in the text and Acknowledgements. This dissertation contains 13,400 words including appendices and tables.

Osman Khawer Zubair
August 2018

Acknowledgements

First and foremost, I would like to thank my supervisor, Dr. Ted Meeds. Throughout the project, Ted has provided me constructive advice, support and valuable feedback. He has always been available for meetings, even at short notice. His guidance and depth of knowledge on the subject has helped me immensely. I am also grateful to Prof. Bill Byrne who gave me constructive feedback and support. My meetings with him always left me with more confidence in my work. I would also like to thank Karen Ullrich, who's original work this project is built upon, for providing me intuition and clarity on the algorithm. I am also grateful to Microsoft for providing Azure credits for experimentation.

Abstract

Deep neural networks have become significantly more powerful and ubiquitous. Used for a plethora of different applications, they can be found in many consumer products and enterprise applications. The rise in computational resources and sizes of datasets has made these networks larger and more complex. However, deeper networks consume more energy and limit applications due to their size.

Compressing these networks while maintaining accuracy is a popular area of research. This project investigates an algorithm for compression, soft-weight sharing, and proposes modifications to improve it. First, we study the effects of incorporating knowledge distillation into the algorithm. Second, we allow each layer to be scaled differently to allow more flexibility for compression. Lastly, we implement layer-wise compression, whereby we mimic each layer of the fully-trained network in hopes of making optimisation easier. Experiments are carried out and analysis is provided for all the modifications.

Table of contents

List of figures

List of tables

Chapter 1

Introduction

Artificial intelligence has enjoyed a resurgence in recent years in large part due to availability of more data and faster computation. Deep learning in particular has successfully been applied to many complex tasks. With deep learning becoming more and more prevalent, we would like to reduce the computational resources required for a variety of different reasons. This project investigates an algorithm for compressing neural networks, soft-weight sharing [24], and modifies it with the aim of further improving compression.

1.1 Motivation

Deep Neural Networks (DNNs) have been successfully applied to a plethora of different tasks in various domains such as computer vision, natural language and speech. An unprecedented rise in both, the computational power in modern hardware, as well as the data collected has allowed for more complex and accurate models to be trained. In order to further increase accuracy, networks are often made larger, in order to find and represent the complex underlying patterns in the data.

Larger DNNs pose significant drawbacks which hinder machine learning applications. A large model requires greater storage capacity and bandwidth when transferred. More processing and memory access cycles are required to carry out the computation if the size of the model increases. This results in slower processing and greater energy usage [3] [6]. Increase in computation complexity and energy usage increases fixed and operating costs. Many models cannot be implemented due to storage, battery and processing constraints. As a result, data is collected from these devices and sent over the network for inference on more powerful devices. The transfer of data introduces not only latency, but can also be a cause for

privacy concerns for consumers.

If we divide computing hardware into 3 broad categories, we can see benefits at each scale. The first is CPUs and GPUs without constraints of energy or space, commonly found as desktops or servers in datacenters. These often contain 16GB or more memory, hundreds of GPU cores and terabytes of storage. The second category is full-featured mobile devices, with constraints of energy usage. This would include mobile phones, smartwatches and single-board computers such as Raspberry Pi. In general, these devices hold 512MB - 3GB of memory and low-powered ARM application processors (A-series). Lastly, we have ultra-low power devices which are mainly used as sensors. This would include wearables such as fitness trackers. Such devices normally have 16-32KB of memory and an ARM microcontroller (M-series).

	Processors	Memory	Storage
1	GPUs	16GB+	>0.5TB
2	Mobile CPUs	512MB - 3GB	<0.5TB
3	Microcontrollers	16KB - 32KB	<1MB

Table 1.1 Computational Devices Categorisation

The first set of hardware is mainly used in desktops or datacenters. While energy is not a constraint per se as these devices are connected to the electric grid, computationally complex programmes contribute to running costs. Furthermore, more hardware is required to match processing demands, increasing fixed costs. A 2016 report estimates datacenters consume 3% of total electricity and produce 2% of global emissions [2]. Using less computation can save energy, emissions and costs at this scale.

In case of full-featured mobile devices, inference can be carried out using smaller neural networks where either the task is not complicated or the neural network has been compressed. With limited storage, memory, energy and computational resources, most deep neural networks cannot be used on these devices directly and input data is transferred to a cloud service. For example, inference for a single image using VGGNet model requires 800MB of memory, and 160ms processing time on a mobile GPU [3]. Transferring input data to cloud increases risk of breaches and privacy concerns and also requires connectivity, restricting certain applications. Amazon Alexa and similar home assistants do not start listening for instructions until a keyword is used but if the device could function without network connectivity and input data remained on-device, it would not be risky for the device to keep listening. The

home assistant can chime in without being prompted and customers would also be more open to allowing webcam or video integration in virtual assistants. There are many other potential applications that can be unlocked by on-device inference.

Ultra-low power microcontrollers cannot generally use complicated statistical models and rely on rule-based models or simple signal processing techniques. The constraints on memory and processing power are too great given current technology, but as we move to a more connected world with the Internet of Things, we would like to use sensors which can run inference on chip and send only the important alerts and information across. For example, biosensors with complex models can be used to track different medical signs and indicate anomalies.

Hence, reducing computation helps reduce energy consumption and costs for large-scale datacenters and unlocks many new applications on smaller devices.

1.2 Contributions and Research Questions

Various techniques have been investigated to compress neural networks. In general, these involve removing the least important parameters from the network, reducing the number of bits needed to represent parameters or using a smaller network to mimic a larger network. The thesis mainly studies and investigates potential improvements to a neural network compression technique, soft-weight sharing.

Soft-weight sharing was initially introduced as a technique for regularisation [16] and later shown to be very effective at compressing networks [24]. The technique involves retraining a pre-trained neural network with a Gaussian Mixture Model (GMM) prior over the parameters to force weights to cluster. The process is described in greater detail later in the thesis. We try to improve upon this algorithm through three potential areas of improvements.

First, we would like to incorporate knowledge distillation into soft-weight sharing and study the effects. Knowledge distillation is a technique whereby we use predictions from a complex neural network, rather than the true labels, as targets to train a smaller or less complex neural network. Using prediction targets, more information is provided to the smaller network than hard labels. This allows smaller and more constrained networks to mimic larger networks. Our first hypothesis is that using knowledge distillation with soft-weight sharing

will improve compression.

Second, we investigate layer-wise modifications to improve soft-weight sharing. When a single GMM prior is shared over the whole network, the densest layers dominate the GMM prior loss and hence the means. Our hypothesis allowing the mixtures means to be multiplied by different scales at each layer can improve accuracy by reducing constraints on smaller layers.

Third, we would like more independence between the different layers of a neural network during compression. Our last hypothesis is that having layer-by-layer compression should further improve sparsity in the network as optimising each layer separately significantly simplifies the optimisation problem to essentially a regression problem. The GMM prior constraints should optimise better if the optimisation for accuracy becomes less complex.

Our three research questions are summarised below:

1. What effects does knowledge distillation have on soft-weight sharing?
2. Can we improve soft-weight sharing by allowing different scales for Gaussian mixtures in each layer?
3. Is it better in terms of compression for a network to be mimicked layer-by-layer or as a whole?

1.3 Thesis Outline

First we need to understand compression within neural networks, what it means and what techniques are currently being used. **Chapter 2** gives a brief overview of neural network architecture followed by a review of various techniques currently being used to compress neural networks.

As soft-weight sharing is a critical part of our thesis, we describe and study this in detail. Similarly, the theory behind knowledge distillation along with potential improvements is also presented in detail. **Chapter 3** provides the theoretical details of the methods.

In **Chapter 4** we present experiments and results to provide intuition into these algorithms and help assess the quality of each of the methods.

Chapter 5 contains a discussion of the results to help answer the research questions.

The thesis in concluded in **Chapter 6** with a brief summary of the work undertaken, results and future areas of development.

Chapter 2

Background

This chapter provides a brief overview of neural networks and deep learning along with a review of the work undertaken in compressing neural network so far. A brief description of all major methods is provided here and chapter 3 will further expand on the techniques being utilised for this thesis.

2.1 Neural Network Architecture

The artificial neural network (ANN) is loosely inspired from biological neurons which are connected and cascaded to perform computation. Similarly, in deep learning a feedforward neural network or multi-layer perceptron (MLP) is composed of separate functions which feed into each other. ANNs act as universal function approximators, mapping an input x to an output y using learned parameters. A single perceptron is shown in figure 2.1 with the output equation.

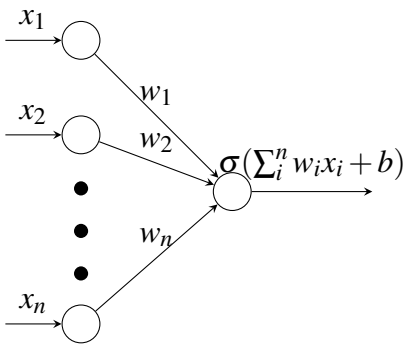
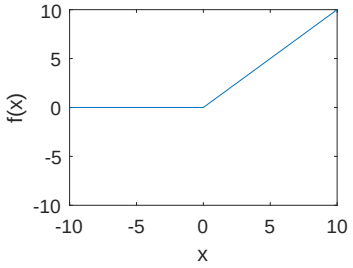


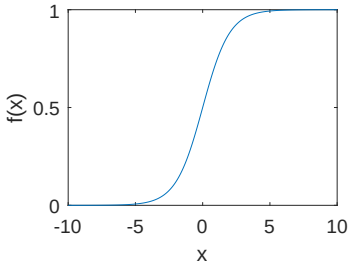
Fig. 2.1 Perceptron

The weights (w_n) and bias (b) in such a network are learned using labelled data. The activation function (σ) is a non-linearity that maps the nodes to the output. The Sigmoid or ReLU (Rectified Linear Unit) functions are common examples of activation functions.



$$f(x) = \begin{cases} x, & x \geq 0 \\ 0, & x < 0 \end{cases}$$

(a) ReLU



$$f(x) = \frac{1}{1+\exp(-x)}$$

(b) Sigmoid

Fig. 2.2 Activation Functions

An MLP consists of input, output and hidden layers. The addition of hidden layers increases representational power of the neural network as cascading and non-linearities result in more flexibility to approximate the underlying pattern in the data. Figure 2.3 shows the diagrammatic representation of an m-class MLP with 2 hidden layers and n inputs.

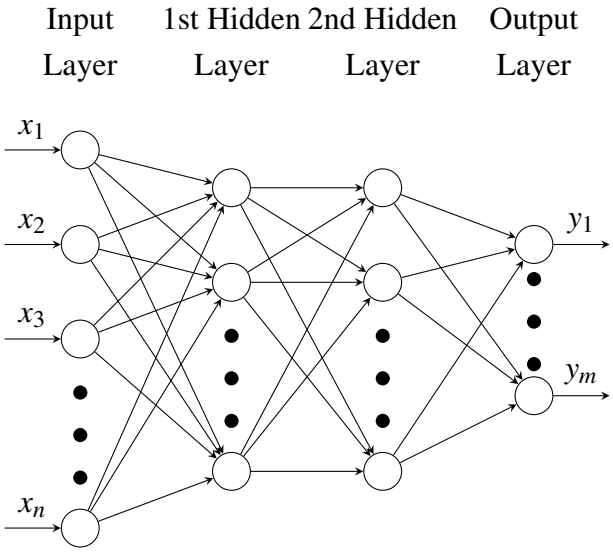


Fig. 2.3 Feedforward Neural Network

Optimization is performed using the training data and labels to train the neural network. A loss function is used to indicate the error in the network's prediction $\hat{y}$ against the training

labels y. This error is backpropagated and the weights in the neural network are adjusted to minimize the error. Cross-entropy loss function is commonly used when the output is a probability between 0 and 1.

$$\mathscr{L}_{CE}(\mathbf{w}) = -\frac{1}{N}\sum_{i=1}^{N} y_i \log(\hat{y}_i) \tag{2.1}$$

For real-valued targets, the mean-squared error loss can be used as a loss-function.

$$\mathscr{L}_{MSE}(\mathbf{w}) = \frac{1}{N}\sum_{i=1}^{N} (y_i - \hat{y}_i)^2 \tag{2.2}$$

2.1.1 Examples of Popular Architectures

A special type of feedforward neural network, the convolutional neural network (CNN) is commonly used for images [11]. CNNs exploit spatial locality in image data and use 3-dimensional neurons for learning. The filters in a convolutional layer are shared over the complete visual field, reducing the number of parameters significantly compared to a fully-connected network. Recurrent Neural Networks (RNNs) are another popular class of neural networks for sequence data [5].

LeNet-5 was the first application of CNNs, with 7 layers and 61K parameters over 431K connections [11]. The network introduced in 1998 and used for handwriting recognition. AlexNet was a much larger CNN with 60m parameters, introduced in 2012. It used ReLU activations to mitigate the vanishing gradient problem and dropout to reduce overfitting [10] [20]. ZFNet made minor improvements to AlexNet with changes to convolutional filter sizes and hyperparameters [27].

GoogLeNet, released in 2014, contains 22 layers [21] but only 4m parameters, as it replaced the dense fully-connected layers with more convolutional layers. Even though the number of parameters is reduced, the number of computational operations per parameter increases due to excessive layering. VGGNet, released the same year, introduced 16 and 19 layer versions with 138m and 144m parameters respectively [19]. ResNet, released in 2015, used skip connections to overcome vanishing gradients [8]. Hence, a 152-layer deep version was successfully trained.

Table 2.1 contains the summary of the networks with top 5 error rate on ImageNet for each. A trend of increase in number of layers and parameters with decrease in error rate

can be seen. Further reduction in error rates can be acheived by ensembling models, which significantly increases the number of parameters.

Network	Year	Parameters	Layers	Top-5 Error Rate (%)
LeNet-5	1998	61K	7	N/A
AlexNet	2012	60m	8	15.3
ZFNet	2013	60m	8	11.7
GoogLeNet	2014	4m	22	6.7
VGGNet-16	2014	138m	16	7.3
ResNet	2015	60m	152	3.6

Table 2.1 ImageNet Neural Network Architectures Summary

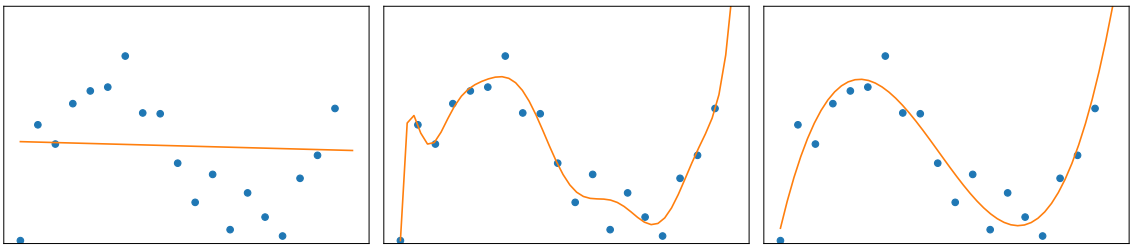
2.2 Compression

Neural network model compression can be approached from different perspectives. The first and most common view is simply the storage representation of a model. This is perhaps the most relaxed and straightforward view of compression as it does not take into account processing at all. Run-time memory usage can also be used to quantify compression, especially if memory complexity of the algorithm is high. Processing complexity can be quantified in floating-point operations or inference time to indicate compression. The latter two would be heavily dependent on hardware and software implementations. We will be approaching compression from the angle of storage as it is the easiest to quantify. All 3 cases generally lead to lower model complexity.

Different criteria exist for model selection which account for model complexity. One such criterion that also focuses on compression is the minimum description length (MDL) principle. MDL is analogous to Occam's razor and states that the best hypothesis, indicated by model and parameters, is one that best compresses data. MDL can also be seen as a variational learning problem. With data $\mathscr{D}$ and parameters w, the variational lower bound can be decomposed into the complexity loss ($\mathscr{L}^C$) indicating the cost to describe the model, and the error loss ($\mathscr{L}^E$) indicating the misfit between the model and data, where $q(w)$ is the approximation of the posterior $p(w|\mathscr{D})$ [24].

$$\mathscr{L}(q(w),w) = -\mathbb{E}_{q(w)}\left[\log\left(\frac{p(D|w)p(w)}{q(w)}\right)\right] = \underbrace{\mathbb{E}_{q(w)}[-\log p(D|w)]}_{\mathscr{L}^E} + \underbrace{KL(q(w)||p(w))}_{\mathscr{L}^C} \tag{2.3}$$

Figure 2.4 gives the intuition of the MDL principle applied to curve-fitting. Excessively low complexity does not allow the model to learn from the data and results in high error. High complexity allows the model to overfit to the data and not generalise well. MDL provides a means to achieve a trade-off and produce a model that performs and generalises well.



(a) Low Complexity (b) Low Error (c) Error-Complexity Trade-off

Fig. 2.4 MDL Model Selection Example

Compressing neural networks has become a popular topic in the machine learning community and much research has been done in this area. Most of the approaches roughly fall into some combination of three approaches:

1. Pruning: Removing the least useful weights and operations in a network, leaving behind a smaller or compressed network.
2. Quantisation: Reducing the number of bits needed to represent the weights in a network.
3. Knowledge distillation or joint training: Training a smaller network to mimic a larger network by transferring the knowledge from the complex network

Each of these are further explained in detail in the following subsections.

2.2.1 Pruning and Sparsity

Quite a few different approaches have been taken in order to prune neural networks effectively. In general, pruning involves finding out which weights and operations are least important and can be removed with minimal impact to accuracy. An interesting analogous

observation is that the number of synapses in human brains increases in the first year but drops off drastically in the second year and continue to decrease into adolescence [25].

Using a Bayesian approach to learning, a prior can be placed over the parameters with higher density towards zero. An L2 penalty essentially imposes a Gaussian prior which pushes weights towards zero. Dropout imposes a Bernoulli prior over the nodes and the network trains without using the deactivated nodes [20]. Although these methods are mainly used for regularisation, they demonstrate how significant portions of a network can be removed without harming accuracy.

The effectiveness of pruning makes one wonder whether networks are overparameterised in the first place and initialising smaller networks achieves a similar result. Li et. al [12] attempt to gauge the difficulty of a machine learning problem by restricting the number of dimensions of the parameter subspace that optimisation can be carried out in. Good solutions with over 90% begin to appear at 750 dimensions for MNIST, fewer than the number of pixels in each image. The authors referred to this figure as the intrinsic dimension and shows that the number stays roughly constant for a given problem even while increasing number of parameters in a network. Starting from scratch, a simple linear classifier would require 785 parameters for MNIST, and attains 88% accuracy [11]. The authors posit that extra parameters allow more coverage of the solution subspace which makes training easier.

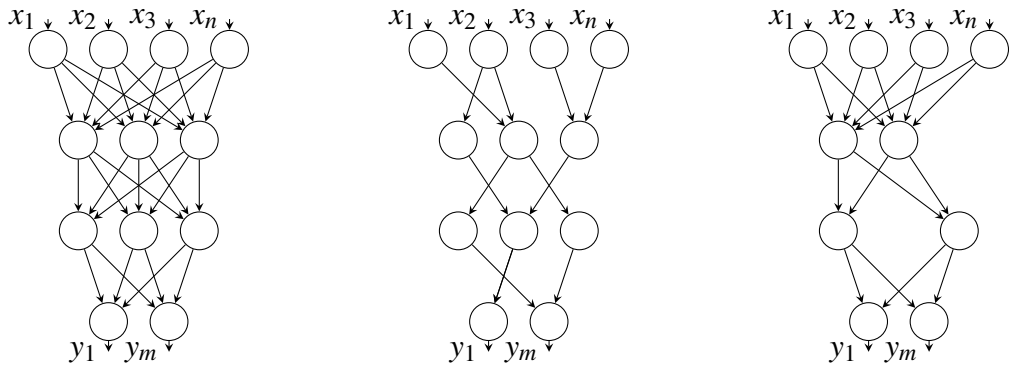
Franklin et. al [4] train a full neural network and pruned it away until a small portion of non-zero parameters remain. They refer to this network as a *winning ticket*. They retrain the winning ticket against a randomly structured network with similar initialisation to the winning ticket. The winning ticket converges faster and achieves higher accuracy, indicating it is easier to find these outperforming networks by training a complex network first and pruning rather than optimising a smaller network. It is unclear whether the added structure allows more flexible representation, or whether our optimisation algorithms are more suited towards deep neural nets or perhaps something else entirely.

A few different methods exist which train full networks and then prune for compression. Soft-weight sharing uses a Gaussian mixture prior over the weights [24], enabling clustering at different Gaussian mixtures. These can be pruned and quantised to mixture means resulting in a negligible accuracy drop. Other approaches utilise variational dropout to introduce sparsity in a deep neural network [15]. In sparse variational dropout a sparsity inducing log-uniform prior is added over the weights. The loss function is modified to include the

KL-divergence between the true posterior distribution and the prior. The method results in over 98% sparsity for LeNet-300-100 on MNIST and is the current state-of-the-art method for compression [15].

Researchers have looked into pruning at different levels of granularity. Figure 2.6 show possible convolutional structures that can be pruned. Figure 2.5 shows connection and node-level pruning in a fully-connected layer. In general, the more regular the pruned structure, the easier it becomes to optimise the model for processing, especially for parallel processing devices like GPUs. However, the restriction also reduces the percentage of parameters that can be pruned away without a considerable loss in accuracy. Mao et. al show for AlexNet on ImageNet, greater than 80% of parameters can be finely pruned without a loss in accuracy but this is reduced to between 60% and 75% when restricted to kernels and vectors [14]. Furthermore, 3D filters could not be pruned without a significant loss in accuracy

Wen et. al [26] propose a learning method which incorporates a group lasso penalty over regular structures to perform processing efficient pruning [7]. Similarly, hierarchical priors have been introduced with variational dropout for group sparsity using variational methods [13]. This project does not consider the hardware processing implications of pruning, and will not attempt for regularity in pruning.



(a) No Pruning (b) Connection Pruning (c) Node Pruning

Fig. 2.5 Pruning in a Fully-Connected Layer

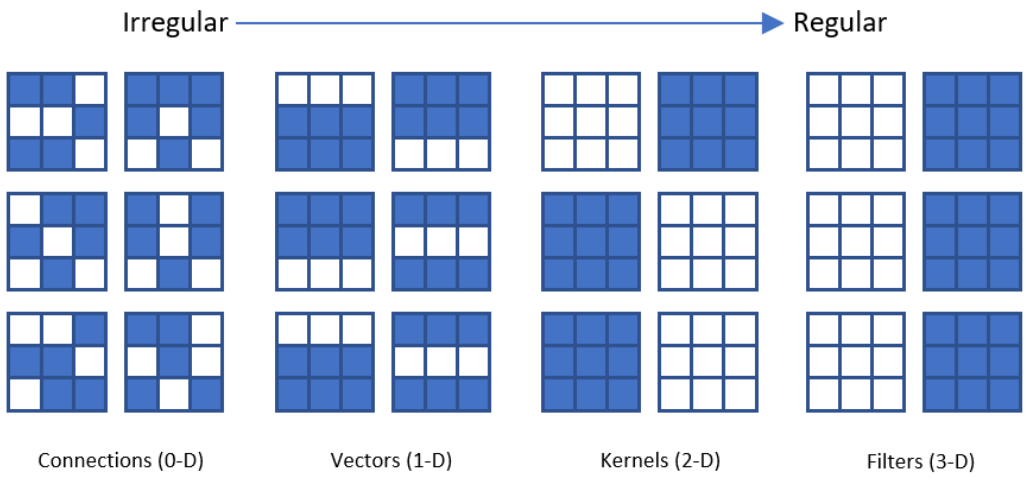


Fig. 2.6 Structure in Pruning [14]

Yet another method to approach pruning is to use iterative pruning [7]. With iterative pruning, the model is trained with an L2 penalty and connections close to zero are deemed unimportant and pruned. Training is re-run with the pruned connections clamped to zero and the network recovers. Iterative prune and retrain allows more sparsity at higher accuracy than one-shot pruning with L2 regularisation.

2.2.2 Quantisation

To reduce the size of a neural network, we can also reduce the number of bits needed to represent the weights inside a network. Normally, these weights are represented as single precision floating points, occupying 32 bits each. Different approaches have been taken for quantisation, with some involving training with quantised weights.

A very straightforward approach is to convert the floating point representation to limited precision fixed point, limiting the dynamic range of the weights but allowing a more compact representation. TensorFlow Lite, a version of the popular deep learning library for mobile devices, allows for 8-bit fixed point quantisation [22]. This immediately reduces the model size down to a quarter of the original size. As a result, less storage and memory access operations are required. Furthermore, floating point arithmetic operations require more steps than fixed point and are normally slower without dedicated hardware. Tests show that through this technique, the drop in accuracy for MobileNet network for ImageNet challenge reduces by a mere 1-2% [22]. This is quite remarkable as it is achieved without any additional

training or optimization unlike many other methods.

Several methods exist which train with quantised or low precision weights. One example is a binarized neural network where the weights are restricted to $-1, +1$. The network is trained with the binary constraint and results show no loss in accuracy for AlexNet on ImageNet challenge even though $32\times$ less memory is required and computation is $\approx 2\times$ faster due to multiplication being replaced by addition and subtraction [17]. Further restricting the input data space to binary values of $-1, +1$, creates a network that can effectively carry out nearly all computation using XNOR logic gates. This results in a $\approx 58\times$ speedup but a $\approx 13\%$ drop in accuracy for AlexNet on the ImageNet.

Extending the idea of binary quantised network further, a ternary quantisation method allows for 3 weights in any given layer, $W_l^n, 0, W_l^p$. W_l^n and W_l^p are negative and positive weights respectively, for each layer. Hence, there are only 2 learnable values per layer. The error backpropagates to the learnable values and also determines which of the values each weight would occupy. The results show a no decrease in accuracy for ResNet-32, ResNet-44 and ResNet-56 on the CIFAR-10 dataset [1].

Further increasing the number of learnable weights, soft-weight sharing uses GMM prior over the weights to cluster them around mixture means. The 0-mean mixture has a fixed mean and a high mixing proportion, while the remaining mixtures' means are free parameters. A pre-trained network is then clustered using the GMMs. Once clustered, the network can be quantised by setting each weight to the mean of the Gaussian mixture it belongs to. The 0-mean mixture is pruned while the remaining are quantised to their means. Codebook quantisation is used for compression. The mixture mean values are stored in a codebook. The index of the codebook, which requires fewer bits, can be used to represent the weight matrix of the model instead of 32-bit floating points [24]. Codebook quantisation is further discussed in section 4.2.4.

2.2.3 Distillation and Joint Training

Knowledge distillation, also commonly referred to as teacher-student training, uses a smaller model (the student) to mimic a larger model (the teacher). This is especially useful in the case of ensemble models but can also be used for deeper neural networks to replace them with shallow ones. Distillation is essentially carried out by using the predictions of a more complex neural network as the training targets for a smaller network [1] [9]. The input

data with the predictions from the complex network form the transfer set for distillation.

Figure 2.7 gives an intuition as to why distillation works using the example of the MNIST handwritten digit recognition task. In the case of normal training, the target labels for each digit are one-hot encoded. Therefore, the network must learn to fit to these 1s and 0s. Using predictions from a more complex neural network provides more information to the smaller neural network for training. In the different training cases shown in figure 2.7, we see that prediction weighting is divided among digits, letting the network learn for all digits. The first training case contains a 3 that resembles an 8, and so gives some prediction weighting to 8. Hinton et. al also show that removing all 3s from the transfer set and slightly tweaking the bias results in 98.6% recognition accuracy for 3 despite the network never having seen a 3 [9]. The softer distribution given by the transfer set can also be modelled by the shallow network more easily as distributions with sharp peaks require more representational power.

Distillation is discussed in more detail in chapter 3 as it forms one of the fundamental building blocks of the thesis.

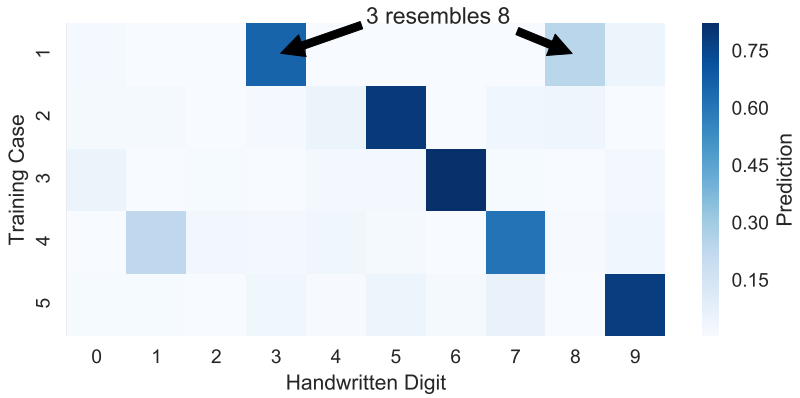


Fig. 2.7 MNIST Distillation Targets

Joint-training uses a slightly different approach to produce similar results. Projection-Net consists of two networks being trained simultaneously; a full-fledged deep neural network and a projection network optimised for inference [18]. The projection network is smaller and mainly contains efficient on-device operations. The projection network mimics the teacher network with larger capacity and they are both trained jointly. Two loss functions are defined, one for the true accuracy of the trainer network and the other for the distillation accuracy in the projection network. Once trained, the two networks are decoupled. The trainer network can be used for inference on more powerful devices, and the projection network can be

transferred to mobile devices.

In the next chapter, we expand upon the methods relevant to our project and provide theoretical details of soft-weight sharing and our modifications.

Chapter 3

Methods

Our project uses soft-weight sharing as a starting point for experimentation and implements potential improvements to the algorithm. This chapter explains the soft-weight sharing algorithm, and our proposed modifications in detail.

3.1 Soft-weight Sharing

Originally, soft-weight sharing was proposed as a technique to improve generalisation and penalize complexity in a neural network. In earlier days of machine learning, labelled data was not as frequent which led to models that did not generalise well. A possible solution to this was a technique called weight-sharing whereby a single weight was shared by multiple connections. Another approach is to include a complexity penalty within the cost function during training. This can be re-conciliated with the MDL view as error being the data-misfit cost and complexity being the cost to describe a model. Similarly, complexity can also be viewed as a prior belief on the appropriateness of the model.

$$\text{Cost} = \text{Error} + \tau\ \text{Complexity} \tag{3.1}$$

An example would be the use of regularisation or weight decay where $\sum_i w_i^2$ is the complexity term which pulls the weights closer to zero with the rate proportional to the weight's magnitude. The error term keeps the network accuracy and also prevents weights from collapsing to zero.

However, the L2 penalty only places a single distribution with a mean of 0. In some cases, this not optimal for reducing complexity. Take the case where two correlated units are

present with weight $\frac{w}{2}$. Instead of preferring to turn one off and doubling the other (i.e. w and 0) the network prefers to keep both of them at their current value as there is lower penalty.

$$\left(\frac{w}{2}\right)^2 + \left(\frac{w}{2}\right)^2 \leq w^2 + 0^2 \tag{3.2}$$

One way to mitigate this is to use a spike-and-slab prior with narrow and broad Gaussian mixtures:

$$p(w) = \pi_n \frac{1}{\sqrt{2\pi}\sigma_n} \exp^{-w^2/2\sigma_n^2} + \pi_n \frac{1}{\sqrt{2\pi}\sigma_b} \exp^{-w^2/2\sigma_b^2} \tag{3.3}$$

The conditional probability that a particular mixture generated a particular weight is given by responsibility:

$$r_j(w_i) = \frac{\pi_j p_j(w_i)}{\sum_k \pi_k p_k(w_i)} \tag{3.4}$$

Weights with higher values will have greater responsibility for the broad mixture, and weights with lower values will show greater responsibility for the narrow mixture. Building upon this, we can implement multiple Gaussian mixtures with learnable parameters. The complexity penalty can help cluster the weights effectively, grouping parameters that can be shared.

$$p(\mathbf{w}) = \prod_{i=1}^{I} \sum_{j=0}^{J} \pi_j \mathcal{N}(w_i | \mu_j, \sigma_j^2) \tag{3.5}$$

The prior is initialised with a fixed number of mixtures (J). The mean (μ_j), variance (σ_j^2) and mixing proportions (π_j) are learned during compression. One component has its mean fixed at zero ($\mu_{j=0} = 0$) and mixing proportion close to 1 ($\pi_{j=0} \approx 1$) to ensure more weights cluster around zero value. To prevent the mixtures from collapsing to single data points, we impose inverse Gamma priors on the precisions of the mixtures. The Gaussian mixtures can be initialised with high variance which can be reduced to push weights closer to mixture means.

The loss function is updated to include the GMM prior loss. The trade-off parameter, τ is used to balance the loss function between accuracy loss and complexity loss. A higher τ would increase the clustering and sparsity in the network at the cost of accuracy and vice versa. Eventually during training the gradients from the two loss penalties, accuracy and complexity, will begin to cancel out, leading to a final optimised state. Cross-entropy or mean-squared error loss function can be modified for use.

$$\mathscr{L}_{CE}(\mathbf{w}, \{\mu_j, \sigma_j, \pi_j\}_{j=0}^{J}) = \underbrace{-\sum_{c=1}^{C} y_c \log(\hat{y}_c)}_{\text{Cross-Entropy Loss}} \underbrace{-\tau \sum_{i=1}^{I} \log \sum_{j=0}^{J} \pi_j \mathscr{N}(w_i|\mu_j, \sigma_j^2)}_{\text{Gaussian Mixture Prior Loss}} \tag{3.6}$$

$$\mathscr{L}_{MSE}(\mathbf{w}, \{\mu_j, \sigma_j, \pi_j\}_{j=0}^{J}) = \underbrace{\frac{1}{N}(y-\hat{y})^2}_{\text{MSE Loss}} \underbrace{-\tau \sum_{i=1}^{I} \log \sum_{j=0}^{J} \pi_j \mathscr{N}(w_i|\mu_j, \sigma_j^2)}_{\text{Gaussian Mixture Prior Loss}} \tag{3.7}$$

Pruning and Quantisation

Once the weights have been clustered around the mixtures, each weight is set to the mean of the mixture that takes the most responsibility. The original implementation [24] also merges the mixtures if KL divergence between two mixtures is below a given threshold. Two mixtures, i and j, and their parameters can be updated as follows:

$$\pi_{new} = \pi_i + \pi_j, \qquad \mu_{new} = \frac{\pi_i \mu_i + \pi_j \mu_j}{\pi_i + \pi_j}, \qquad \sigma_{new}^2 = \frac{\pi_i \sigma_i^2 + \pi_j \sigma_j^2}{\pi_i + \pi_j} \tag{3.8}$$

However, this introduces another hyperparameter which requires tuning and the merging of mixtures does not help with compression unless the number of mixtures is at least halved which would reduce the number of weight encoding by 1 bit per weight (see section 4.2.4). In our implementation, we do not merge mixtures.

The complete algorithm for soft-weight sharing is presented in the listing Algorithm 1.

Algorithm 1 Soft-weight Sharing for Neural Network Compression (adapted from Ullrich et. al [24])

Input: Pre-trained Neural Network

Output: Sparse Neural Network with Quantised Parameter Values

1. Set Hyperparameters:

$\tau \leftarrow$ Trade-off between accuracy and complexity loss

$\eta \leftarrow$ Learning rates for weight parameters and GMM prior parameters

$\mu, \sigma \leftarrow$ Configure inverse Gamma prior distribution on mixture precision

2. Initialise:

$\mathbf{w} \leftarrow$ set network weights to pre-trained network weights

$\theta = \{\mu_j, \sigma_j, \pi_j\}_{j=1}^{J}$ initialize mixture parameters

3. Cluster:

while w, θ *have not converged* **do**

  $\mathbf{w}, \theta \leftarrow \nabla_{w,\theta}\mathscr{L}^E + \tau\mathscr{L}^C$ update parameters with error and complexity gradients

end

4. Prune and Quantise:

$\mathbf{w} \leftarrow \text{argmin}_{\mu_k}(\mathbf{w} - \mu_k)$ set parameter values to closest mean

3.2 Knowledge Distillation

Knowledge distillation or teacher-student training, attempts to make a shallow student network mimic a complex teacher network by using predictions from the teacher network as targets. This allows a higher accuracy to be attained by a smaller network than would be possible with normal training. A brief overview on distillation has already been presented in section 2.2.3.

Initial work on distillation extracted logits (z_i) from ensemble models and tried to have shallow networks match the logits to mimic the complex network [1]. A mean-square error loss was used between the two logits, with a trainable β parameter.

$$\mathscr{L}_{KD}(\beta, \mathbf{w}) = \frac{1}{2N}\sum_{i=1}^{N}(\beta z_i^{(S)} - z_i^{(T)})^2 \tag{3.9}$$

A novel idea later presented in knowledge distillation is “dark knowledge”, which posits that there is useful information in the tail-end of the output distribution [9]. The final layer of a neural network typically converts the logits (z_i) to a probability using a softmax (q_i).

Softmax pushes the highest value close to 1, and the remaining close to 0, resulting in hard targets. Using common loss functions such as cross-entropy or mean-squared error, the penalty for not matching the high-valued output with the correct label is significant, but the remaining outputs in the tail-end produce negligible error gradients. To increase the impact of the tail-end, we can train by directly using logits as the targets, before the softmax. Another alternative is to use soft-targets which can be produced by dividing the logits by a common value, called the temperature (T) and then carrying out the softmax operation:

$$q_i = \frac{\exp(z_i/T)}{\sum_j \exp(z_j/T)} \tag{3.10}$$

Figure 3.1 shows the effects of changing the temperature on targets. A higher temperature pushes the tail-end further up and produces a smoother distribution. Hence, a smaller temperature corresponds with greater information in the distribution, if viewed from an entropy perspective. Greater variance in the temperature results in more constraints on the weights to model the distribution. A higher temperature produces a more uniform distribution, with less information. Taken to the extreme, modelling a perfectly uniform distribution should only require a single bias parameter. Effectively, this should result in a trade-off between accuracy and sparsity. Using soft-targets can minimize the effects of mislabelled data in the training set and can also allow training without actual data as synthetic data can be used to produce targets with the teacher model.

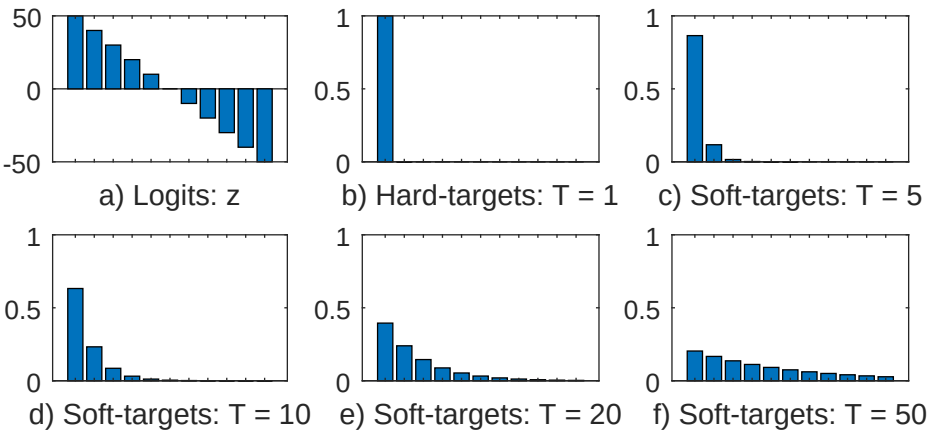


Fig. 3.1 Effects of Changing Temperature on Soft-Targets

Our hypothesis is that using these soft targets can help achieve further sparsity in the compressed network produced by soft-weight sharing as it is easier to model a smoother distribution. Whereas normally, the teacher network is much deeper and the student network is shallower, we use the same network architecture for both student and teacher but have

additional compression constraints on the student network. Additionally, using the transfer set and changing our loss function to use soft-targets will zero out any initial error as our network will already be configured to produce those predictions, effectively cancelling some accuracy loss which may have discouraged weights from clustering.

Different loss functions can be used with knowledge distillation. The softened targets ($q_i^{(T)}$) can be used for training on their own or a dual-objective function with both true labels (y_i) and soft-targets can be used. A cross-entropy loss function with α trade-off parameter between the two is suggested in the paper, with a T^2 multiplier for the soft-targets as gradients scale in an inverse-squared relationship with the temperature (shown in appendix A).

$$\mathscr{L}_{KD}(\alpha, \mathbf{w}) = \alpha T^2 \underbrace{\left[-\frac{1}{N} \sum_{i=1}^{N} q_i^{(T)} \log(\hat{y}_i^{(S)}) \right]}_{\text{Soft-target Loss}} + (1-\alpha) \underbrace{\left[-\frac{1}{N} \sum_{i=1}^{N} y_i \log(\hat{y}_i^{(S)}) \right]}_{\text{True label Loss}} \tag{3.11}$$

Other loss functions, such as the KL-divergence between the student and teacher distributions have also been used for distillation. Our implementation only uses soft-target losses as using hard-targets adds the α parameter, requiring tuning, and adds further complexity to backpropagation. The loss function remains similar to soft-weight sharing (eq. 3.6 and 3.7) but with a T^2 multiplier for scaling accuracy gradients. Another potential benefit of using only soft-targets is that training data may not be required at all as target labels are discarded in the transfer set. Instead of using actual training input data, synthetically generated input could be used. Hence, a model can be compressed without access to original data and this could be beneficial if there are privacy or security restrictions on the original data.

3.3 Layer-wise Scaling

When using L2-regularisation or similar priors over weights which force clustering, we notice that the denser layers cluster more. Normally, neural networks are designed with denser input layers to extract features and further layers condense information. Hence there is greater flexibility in initial dense layers and the spread or the standard deviation can be reduced without hurting accuracy more so than condensed layers.

Another modification we would like to examine is whether allowing the mixtures in each layer to have different scales will improve the compression. The underlying belief this

modification attempts to model is that each layer has a different spread of weights. We allow relative scaling of weight values by allowing a scaling multiplier for each layer. The scale of first layer is fixed to 1 and hence the mixture means in this layer equal the GMM prior means. The remaining layers will have scaling multipliers as free parameters.

Scaling should allow priors to be more flexible for each layer, encouraging compression. In terms of storage and codebook quantisation, only 1 32-bit value is required to be stored per layer which is a small cost.

3.4 Layer-wise Compression

The final area of potential improvement we would like to explore is carrying out compression layer-wise. Our hypothesis is that if we use the transfer set from knowledge distillation and attempt to mimic each layer separately instead of the final predictions at the output of the network, it should lead to greater compression as single layer models are less complex and easier to optimise.

Ideally, we would prefer for each of the layers to be compressed completely independently so that the workload to compress a deep neural network can be split across multiple computational devices. This is visualised in figure 3.2. Computation is also simplified as each layer is simply carrying out logistic regression. However, this would also require each layer to have separate priors, which would increase the number of weights that need to be stored in the codebook and the bits-per-weight required for storing the weight matrix (see section 4.2.4). Another drawback is that error from the final layers is not backpropagated to initial layers at all. Relying on only the local error can be unfeasible as the error may amplify in the layers ahead and this would not be registered in the local loss function.

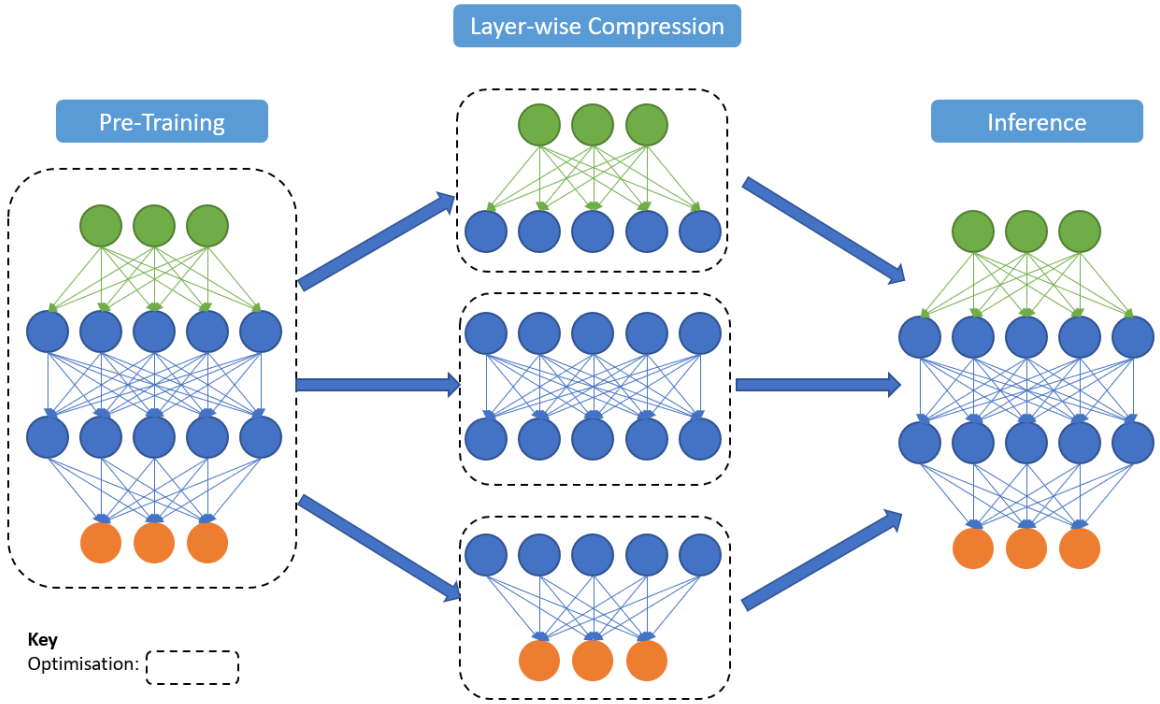


Fig. 3.2 Independent Compression

Another approach is to share the prior amongst all layers and use a loss function that accumulates losses for all layers. A mean-square error loss function can be defined which penalises the prediction at each hidden layer to keep it as close to original prediction as possible:

$$\mathscr{L}_{MSE}(\mathbf{w}, \{\mu_j, \sigma_j, \pi_j\}_{j=0}^{J}) = \underbrace{T^2 \sum_{l=1}^{L} \frac{1}{n_l}(y_l - \hat{y}_l)^2}_{\text{MSE Loss for all Layers}} - \underbrace{\tau \sum_{i=1}^{I} \log \sum_{j=0}^{J} \pi_j \mathscr{N}(w_i|\mu_j, \sigma_j^2)}_{\text{Gaussian Mixture Prior Loss}} \tag{3.12}$$

In this manner, a single GMM prior is shared over the whole network and error is also backpropagated, ensuring any small error does not amplify and cause a large drop in accuracy. At the same time, the layers are being mimicked at each level.

In the next chapter, we implement our methods to verify whether out hypotheses are correct.

Chapter 4

Experiments

In this chapter, we discuss the implementation of soft-weight sharing and the suggested modifications. Results from the experiments are presented along with analysis. Chapter 5 will report and provide discussion on the outcomes of the experiments in view of the research questions.

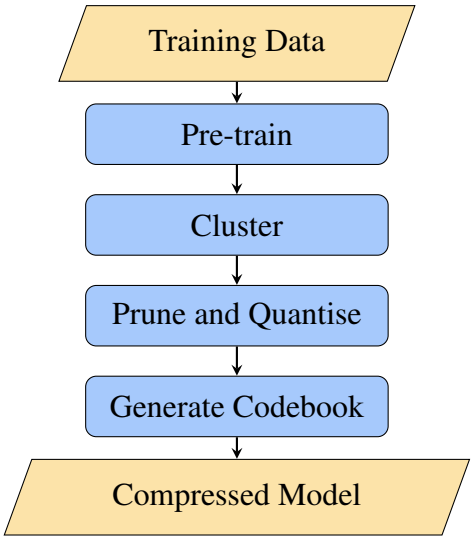


Fig. 4.1 Experimental Setup

4.1 Setup

This section describes our experimental setup and the networks and the machine learning tasks compression will be tested on.

A demonstration code for soft-weight sharing has been publicly shared by original authors in Keras [23]. Our implementation uses PyTorch to make use of its automatic differentiation feature which makes customisations simpler. Most of the soft-weight sharing code was rewritten but the demo code was extremely useful as it provided clear logic and rough hyperparameter values to begin testing.

4.1.1 Dataset

The MNIST handwriting recognition task is popularly used for building and comparing machine learning models. The dataset consists of the 60,000 training and 10,000 test 28 $\times$ 28 pixel examples of the 10 digits. The task is to classify which one of the 10 digits the handwritten scan belongs to. Examples from the dataset are shown in figure 4.2. For hyperparameter tuning, the last 10,000 examples of the training set were used as the validation set and the results presented use the complete training set after tuning.

Fig. 4.2 Image Examples from the MNIST Database

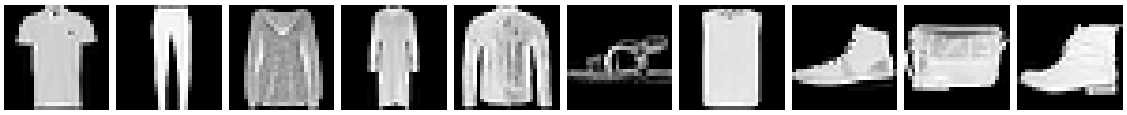


Fig. 4.3 Image Examples from the FashionMNIST Database

While any experimentation, searches and tuning are to be carried out on MNIST, the FashionMNIST dataset is used to verify some results and compare different methods. FashionMNIST is similar to MNIST, in that it contains 28 $\times$ 28 pixel images belonging to one of ten categories that need to be classified. However, the images are different clothing items and harder to classify compared to handwritten digits. Figure 4.3 shows examples from the FashionMNIST dataset for all categories.

4.1.2 Model Architecture

Most of our experimentation is carried out on the LeNet 300-100 architecture. This neural network consists of 2 hidden layers with 300 and 100 hidden units. There are 266,610 parameters contained in the network in total. The LeNet 300-100 is not considered very deep by today's standards but we expect the experimental results to be indicative of compression on larger neural networks. Ideally, we would like to carry out compression directly on much larger networks where the results should be more pronounced. The LeNet 300-100 was used due to its practicality as time and computational power are limited.

Layer	Type	Activation	Dimensions	Parameters
1	Fully Connected	ReLU	$784 \times 300 + 300$	235500
2	Fully Connected	ReLU	$300 \times 100 + 100$	30100
3	Fully Connected	Softmax	$100 \times 10 + 10$	1010
Total				266610

Table 4.1 LeNet 300-100 Architecture

To verify some final results and strengthen our conclusion, a custom convolutional net is used. The architecture, hereafter referred to as Custom CNN, has been taken from the publicly available demo code of soft-weight sharing[23]. This network has 4 layers, of which the first 2 are convolutional and the remaining 2 are fully connected.

Layer	Type	Activation	Dimensions	Parameters
1	Convolutional	ReLU	$1 \times 25 \times (5 \times 5) + 25$	650
2	Convolutional	ReLU	$25 \times 50 \times (3 \times 3) + 50$	11300
3	Fully Connected	ReLU	$1250 \times 500 + 500$	625500
4	Fully Connected	Softmax	$500 \times 10 + 10$	5010
Total				642460

Table 4.2 Custom Convolutional Network Architecture

4.2 Compression Pipeline

The compression pipeline describes the complete stages from training to creating a compressed storage representation of a neural network. The input for the pipeline is simply the training dataset and the output is a codebook quantised parameter matrix for the model.

The pipeline to compression is divided in 4 sections:

1. Pre-training: Original training of the neural network with no complexity penalty
2. Clustering: Implementation of soft-weight sharing and other modifications to enable clustering of weights
3. Pruning and Quantisation: Setting Gaussian mixtures to their mean values, resulting in pruning for the zero-mean mixture and quantisation for the rest
4. Codebook Generation and Compression Rate Calculation: Generate the codebook and compressed representation of parameter matrices in the neural network and determine the ratio of compression

4.2.1 Pre-training

The pre-training step simply produces a trained model that we want to compress. A cross-entropy loss function is used here without any regularisation for 100 epochs. The LeNet-300-100 model reaches **98.26%** test accuracy on the MNIST dataset with this configuration. The fully trained model's weights are clustered in the next step.

4.2.2 Clustering

The soft-weight sharing algorithm is used here to force the parameters to cluster towards one of Gaussian mixtures. This is by far the most computationally complex and time-consuming part of processing as error from two separate loss functions is backpropagated.

Hyperparameters

Many different hyperparameters are involved at this stage making the search for the pareto front incredibly complex. The original paper searched over 13 hyperparameters with 18,000 updates of Bayesian Optimization and still posited the search space may not have been explored well enough [24]. The main hyperparameters involved are listed below:

- τ: Trade-off between accuracy and complexity loss functions (see eq. 3.1, 3.6, 3.7). If the τ value is too low, the model will not remain accurate. On the other hand a low value will form weak clusters with large spreads and drop in accuracy if pruned and quantised to means.

- **Temperature**: The temperature is used to soften the prediction distribution (see 3.1). A softer distribution is produced by a higher temperature which should be simpler to model but reduces distinction between different outputs. A lower temperature produces sharp distributions but loses dark knowledge from the tail-end.

- **Learning Rates**: Multiple learning rates are specified at this step. In normal training, a learning rate is only required for the weights and the weights are only affected by accuracy loss. In soft-weight sharing, the weights are affected by accuracy and complexity loss functions, hence the parameter learning rate becomes more sensitive. Furthermore, the GMM prior has means, mixing proportions and precisions as free parameters, each with separate learning rates. Due to constraints of time and computation, learning rates in the publicly available implementation of soft-weight sharing [23] were used with minor tuning. The layer-wise mixture scaling modification adds another free parameter with a learning rate.

- **Inverse Gamma Prior on Gaussian Mixture Precisions**: To prevent any of the Gaussian mixtures from collapsing to single data points, an inverse Gamma distribution prior is imposed upon the mixtures. The mean and variance needs to be specified for mixture precisions, with higher means encouraging clusters with smaller spread. Furthermore, we define different distributions for zero and non-zero components. The inverse Gamma prior mean for zero-mixture component is kept higher as we expect most elements to fall within this, and would prefer stronger clustering.

In order to ensure a fair comparison between the different methods, hyperparameter searches are carried out to find optimal parameters for each method. Initial attempts at using constrained bayesian optimization and random searches using low-discrepancy sequences did find some optimal points, but finding the pareto front was difficult due to limited compute. Instead, the hyperparameter searches were carried out using either low-dimensional grid searches or hand-tuning. While the priors on mixture precision, loss function trade-off

parameter and temperature were searched, learning rates were only coarsely tuned due to limited computational resources.

Initialisation

The means, variances and mixing proportions of the Gaussian mixtures need to be initialised before they are optimised as free parameters. The zero-mean mixtures is fixed and the remaining means are distributed over -1.0 to +1.0, as this is the approximately the range of values of the pre-trained model weights. Variances are initialised to 0.25. Mixing proportion of the zero-component are set to 0.99, with the remaining 0.01 equally distributed amongst the remaining mixtures. Layer-wise multiplier scaling, where used, is initiated as the standard deviation of each layer relative to the first layer.

4.2.3 Pruning and Quantisation

Once the parameters have clustered into Gaussian mixtures, we quantise them to their means. The zero-mean mixture is effectively pruned leading to a sparse model. The original implementation computed responsibilities for each mixture and merged mixtures if the KL-divergence between them was low. This added a threshold hyperparameter which worsen the already complex search. The mergers do not improve compression either unless mixtures are reduced to a lower multiple of 2 reducing the number of bits per weight needed for representation (e.g. 9 and 16 mixtures require same number of bits per weight for representation). Another downside is the low mixing proportion mixtures are effectively unused. Setting the parameters to their closest mean was found to be more robust during experimentation and used as the method for quantisation.

4.2.4 Codebook Quantisation

Quantised and pruned weights can be represented using compressed storage matrices. At the end of our compression process, we should have pruned the majority of the connections in the network and have the remaining weights set to small number of quantised values, i.e. the means of each of the GMM mixtures. We want to be able to compress and store these weights efficiently and also generate a compression rate metric for comparison.

The method used to compress weights is originally given in Han et. al [6]. A slight modification is applied for practicality. In the original paper a compressed sparse column (CSC) format matrix is used which generates data, column index and recursive index pointer array which keeps a relative count of non-zero elements in each row. Due to presence of multiple layers and 3-dimensional convolutional matrices, we reshape the model into a vector before compressing it, hence only generating the data and column index array as there is only one row.

	Number of Elements	Bits per Element
Original Network	$\#[W]$	32
Weight Codebook	$\#[M]$	32
Scale Codebook (if enabled)	$\#[S]$	32
Weights	$\#[W \neq 0]+$ zero-padding	$\lceil \log_2(\#[M]) \rceil$
Index	Empirically determined	$\approx 6-9$

Table 4.3 Compressed Storage Vector Sizes

Table 4.3 gives a summary of vector sizes involved in compression rate calculation. $\#[M], \#[S], \#[W]$ are the number of mixtures, layer scales and weights respectively. Originally, the storage required for the network is calculated as the number of parameters multiplied by 32-bits for single-precision floating points. We generate a quantised codebook which stores the GMM mixture means as single-precision floating points, allowing us to use low precision integers to map to the floating points instead. Similarly, if layer scaling is enabled, the scales are also stored as floating points, occupying 32 bits each. We then generate the weight matrix which stores all the non-zero weights and place-holder zeroes, representing them by an integer according to the codebook mapping. The bits occupied per non-zero weight depends on the number of mixtures means, scaling as the ceiling of $\log_2$ with the number of mixtures.

Lastly, an index needs to be generated to indicate the position of each non-zero weight. Instead of storing an absolute index position of each non-zero weight, a relative index is used as it requires less bits per element. With an absolute index, a meagre neural network such as the LeNet 300-100 with 266K parameters would require 19 bits per element. Instead, if we use a relative index, we can use a lower number of bits depending on how far we expect consecutive non-zero weights to be. Furthermore, if a gap is too large, we can store a place-holder 0 as demonstrated in figure 4.4 where two consecutive weights cannot be

more than 64 spaces apart due to 6-bit representation of relative indices. Depending upon the sparsity and the relative spread of the non-zero weights of the compressed network, the optimum number of bits to use for the relative index can change.

Absolute Index	10	45	128	145	...
Non-Zero Weights	2	4	1	3	...

↓

Absolute Index	10	45	**99**	128	145	...
Relative Index	10	35	**64**	29	17	...
Weights	2	4	**0**	1	3	...

Fig. 4.4 Using 0 as a Place-holder for 6-bit Relative Index

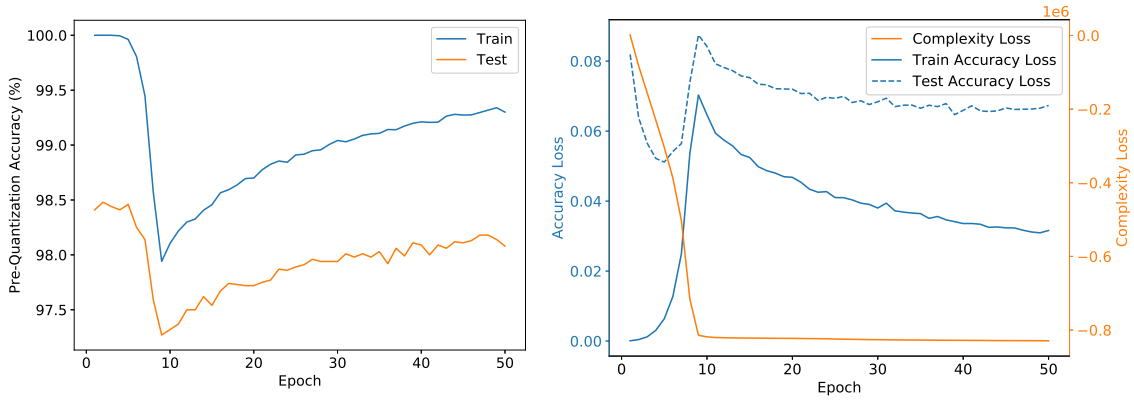
To calculate the compression rate, we simply divide the size of the original network with the combined size of the weights, relative index and codebooks. The compressed representations are generated in code as they are required to determine the size of the weight and index vectors. The compression rate provides a metric that takes both pruning and quantisation into account, whereas network sparsity only accounts for pruning.

4.3 Results

This section presents the results of our experiments along with analysis. Further discussion and analysis takes place in the next chapter in view of the research questions.

4.3.1 Soft-weight Sharing

Initially we tested the pure soft-weight sharing algorithm, without any modifications. Figure 4.5 shows the plots for accuracy and loss functions. The accuracy loss is the cross-entropy on the labels and the complexity loss is from the GMM prior. At epoch 10, we see a cliff where the accuracy falls and so does the complexity loss. This indicates that the weights have moved sufficiently to cluster within the mixtures. 10th epoch onwards, the accuracies begin to gradually rise while the complexity loss stays nearly constant. This behaviour can be thought of as analogous to dropout whereby constraints are placed on the network and it tries to recover and gain accuracy in the constrained state.



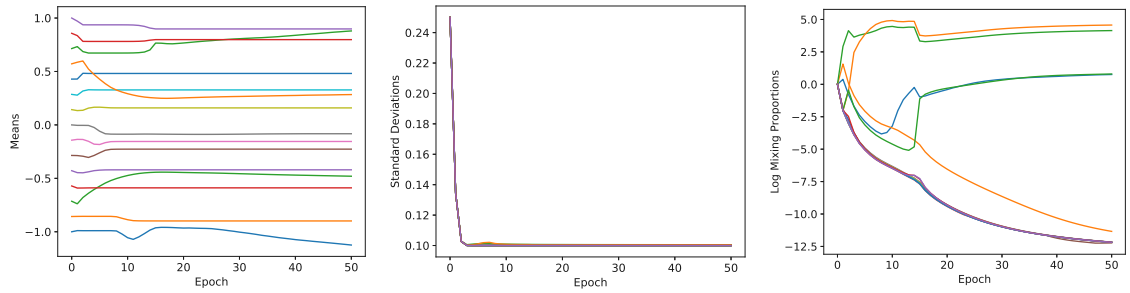
(a) Accuracy

NB: Before Quantisation and Pruning

(b) Loss

Fig. 4.5 Optimisation Curves for Soft-weight Sharing

The free parameters for each of the non-zero Gaussian mixtures can be seen in figure 4.6. The zero-mean mixture only has a free parameter for precision as $\mu_{j=0} = 0$ and $\pi_{j=0} = 0.99$ are fixed. We see that most means stay roughly near where they are initialised. Some mixtures move around quite significantly as they capture more and more mass. The inverse Gamma prior on the mixture precision has a mean of 100, hence the standard deviation falls to 0.1, ensuring better clustering. We see from the mixing proportions that a few mixtures gain most of the mass while most lose mass almost completely as clustering continues.



(a) Means

(b) Standard Deviations

(c) Log Mixing Proportions

Fig. 4.6 Free Parameters for Gaussian Mixtures in Soft-weight Sharing

The model outputs of the first three steps of the compression pipeline can be seen in figure 4.7. The histograms are log-scaled and show the distributions of weight values in the entire network. The pre-training histogram has a roughly Gaussian distribution with a mean at ≈ 0. The clustering pushes the zero-mixture values much closer to enable pruning, however the precision on the non-zero mixtures is lower and there is greater spread. The

plot here shows each of the mixture means as a dark blue line and the shaded area represents 2 standard deviations. We set all the values to the closest mixture means as we prune and quantise.

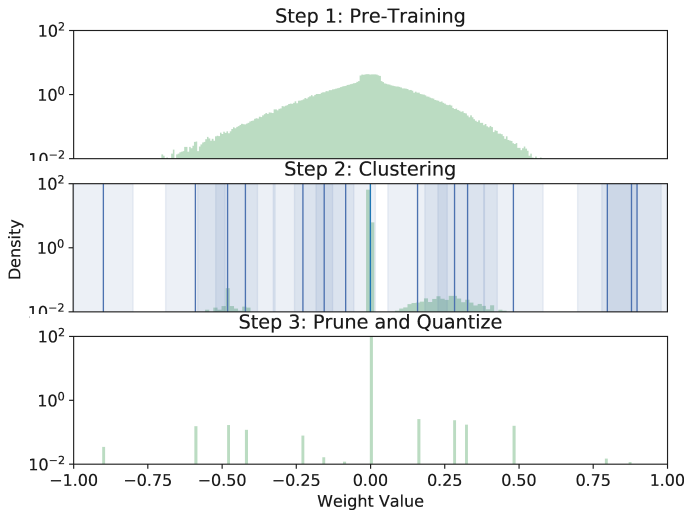


Fig. 4.7 Weight Distributions Changes During Soft-weight Sharing

We can map the movement of parameters during clustering by comparing the pre-trained and clustered distributions. Figure 4.8 shows a joint-plot of weight distributions before and after clustering. The horizontal lines indicate mixture means with 2 standard deviations. At epoch 0, we can imagine the data points being distributed on a diagonal $y = x$ line. The results are given for epoch 50 and majority of the mass from -0.5 to +0.5 has been pulled into zero. The remaining parameters have deviated from their position to fall closer to means of a mixture. A joint-plot for each layer is provided in appendix B.

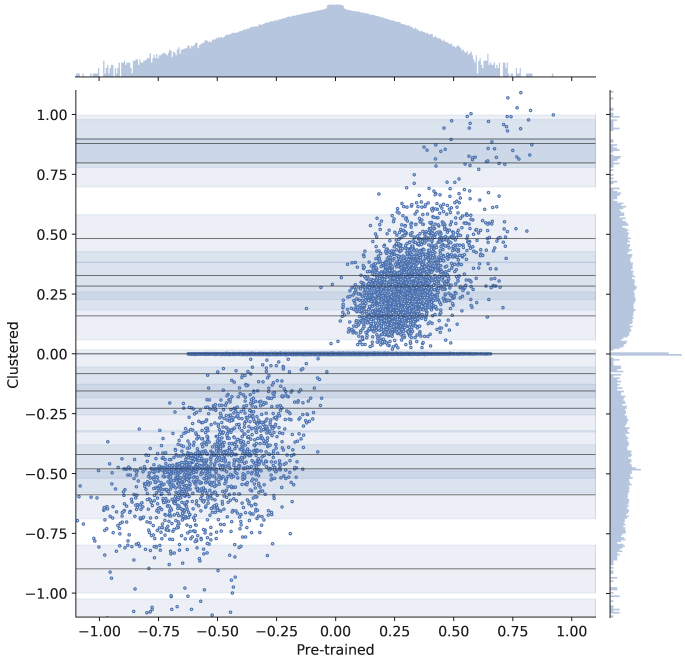
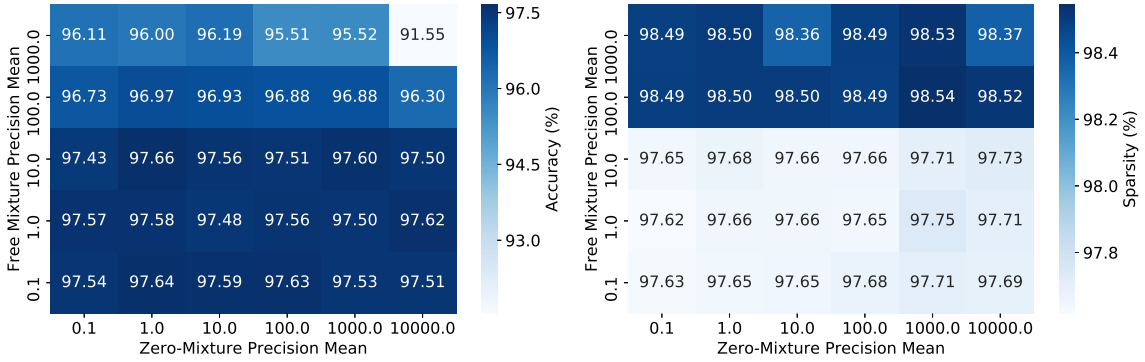


Fig. 4.8 Soft-weight Sharing Distribution Joint-plot

Inverse Gamma Prior

We investigated the hyperparameters for the inverse Gamma prior on precision of mixtures. A mean and variance is defined for the zero-mean mixture and another mean and variance for the remaining non-fixed mixtures. Figure 4.9 shows two heatmaps which report the accuracy and sparsity over a grid of mean values. In general the algorithm is fairly robust to zero-mixture precision mean, as this has a much higher mixing proportion and easily captures mass. However, if the precision for the free mixtures is too high, it becomes harder for them to capture mass and more parameters head to zero. This leads to slightly higher sparsity at the cost of accuracy as seen in the plot.



(a) Accuracy (b) Sparsity

Fig. 4.9 Grid Search for Inverse Gamma Prior Mean for Mixture Precision

Number of Mixtures

The effect of specifying different number of Gaussian mixtures for soft-weight sharing is shown in figure 4.10. We would like to keep the number of mixtures low, as it reduces the number of bits required per weight for codebook quantisation. However, an extra bit is only consumed for increasing mixtures by an order of 2. We see in the figure at 4 mixtures or less, the algorithm does not perform better than random selection. After 6 mixtures we achieve 80%+ accuracy and at 10 mixtures we achieve 97%+ accuracy. Lower number of mixtures slightly favour more sparsity as there are few non-zero mixtures to capture weight values, but this comes at the cost of accuracy. In general, 16 is an appropriate number of mixtures for current setup as it achieves nearly the uncompressed accuracy whereas 8 mixtures still suffer a 3% drop.

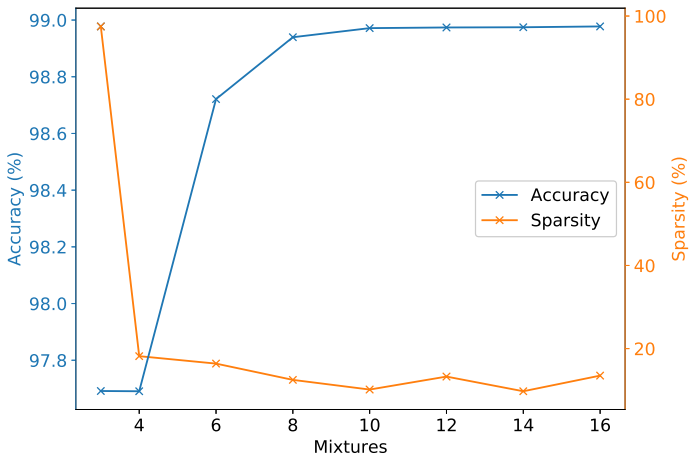


Fig. 4.10 Soft-weight Sharing with Varying Number of Gaussian Mixtures

τ Trade-off Parameter

τ is the trade-off parameter between the accuracy and complexity loss (see eq. 3.1). Increasing τ increases the weighting given to the GMM prior loss. Figure 4.11 shows how increasing τ increases the sparsity at the cost of accuracy. We can vary the accuracy between 97% and 70% for sparsity between 95.5% and 99.5%. Hence, it allows us to easily build either more compressed or more accurate models, depending upon the requirements.

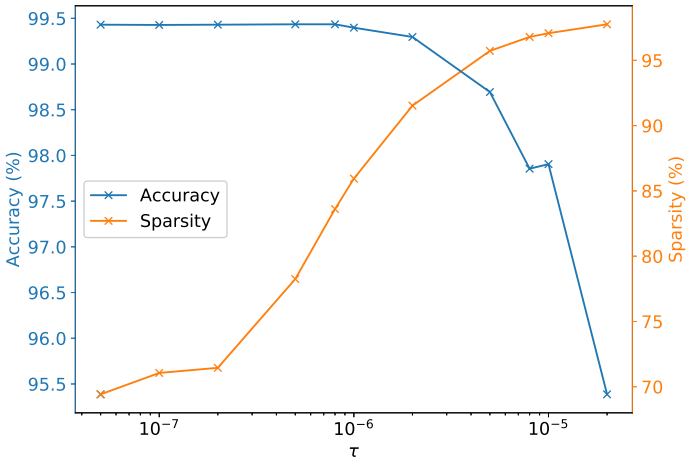


Fig. 4.11 Soft-weight Sharing with Change in τ

Summary Results

Breaking the compression down by each layer, we see that the densest layer achieves the most sparsity in both model architectures as shown in table 4.4. Intuitively, the fewer the parameters in a particular layer, the more information it should contain per parameter. Hence, dense layers have more flexibility to remove parameters without harming performance. The LeNet-300-100 after compression contains 6010 parameters whereas the custom CNN contains 5422 parameters but still achieves higher performance. The CNN has more representational power due to an extra layer and convolutional features, allowing better performance. This finding further supports the idea presented early on that it is easier to train a larger model and compress to achieve a better accuracy than training a smaller model ground-up.

Layer	#[W]	(%)	#[W≠0]
FC1	235500	**98.46**	3625
FC2	30100	94.13	1767
FC3	1010	38.81	618
Total	266610	97.75	6010

Layer	#[W]	(%)	#[W≠0]
Conv1	650	49.54	328
Conv2	11300	88.65	1283
FC1	625500	**99.50**	3144
FC2	5010	86.69	667
Total	642460	99.16	5422

Table 4.4 Layer-wise Sparsity on MNIST with Soft-weight Sharing

Left: LeNet-300-100. **Right:** Custom CNN

Alongside MNIST, the compression was also tested on the FashionMNIST dataset. Whereas MNIST achieves $\approx 99\%$ accuracy on our pre-trained models, FashionMNIST achieves $\approx 90\%$. The FashionMNIST proves to be a harder problem as not only was the original network unable to achieve the same accuracy but the compression also results in greater accuracy drop and slightly lower compression. The LeNet-300-100 faced a 3% greater accuracy drop on FashionMNIST than regular MNIST. The Custom CNN performs much better in this case. On MNIST, the CNN achieves 99.2% sparsity for 0.8% drop in accuracy, and on FashionMNIST it achieves 98% sparsity for a 1.4% drop in accuracy.

Model	MNIST				FashionMNIST			
	OAC	AC	SP	CR	OAC	AC	SP	CR
LeNet-300-100	98.3	97.5	97.8	92.3	89.6	85.8	97.4	86.3
Custom CNN	99.0	98.2	99.2	183.7	89.9	88.5	98.0	150.6

Table 4.5 Soft-weight Sharing Compression Summary Results

Key: *OAC: Original Test Accuracy (%), AC: Compressed Test Accuracy (%), SP: Compressed Sparsity(%), CR: Compression Rate*

The LeNet-300-100 can be compressed $92.3\times$ for storage by using codebook quantisation with 6 bits per index value. Table 4.6 shows the breakdown of how space is used amongst the different arrays in the compressed storage format for the MNIST, with the relative indexing array using the majority of the space.

Array	Space Occupied (%)
Codebook	0.5
Weights	39.8
Index	59.7

Table 4.6 Space Occupied as a Percentage for Each Array in Compressed Format

This section has fully explored the original soft-weight sharing algorithm and shown that it is extremely effective at compression. The next sections examine modifications to soft-weight sharing by carrying out similar experimentation.

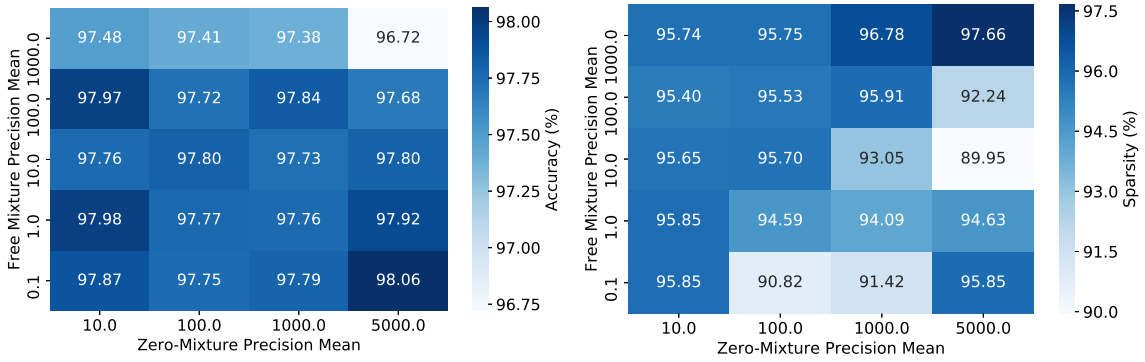
4.3.2 Knowledge Distillation with Soft-weight Sharing

This section details the experiments carried out on soft-weight sharing using predictions as targets rather than the hard-labels originally used for training and clustering. Using distillation, we see a smaller neural network can learn to mimic a larger one, but we would like to see if the same neural network's constraints can be relaxed to allow more sparsity under soft-weight sharing.

The algorithm for soft-weight sharing is the same after incorporating knowledge distillation, the only difference is the use of a transfer set with prediction targets rather than the original training set with hard labels.

Inverse Gamma Prior

Some hyperparameter searches were carried out to study their effects with knowledge distillation. Figure 4.12 shows the effect on accuracy and sparsity for different inverse Gamma prior means on Gaussian mixture precisions. The accuracy stays relatively fixed around 97% but the sparsity can vary somewhat significantly, especially as the zero-mixture mean is increased. Overall, it shows more sensitivity than pure soft-weight sharing which likely due to softer target constraints in knowledge distillation.



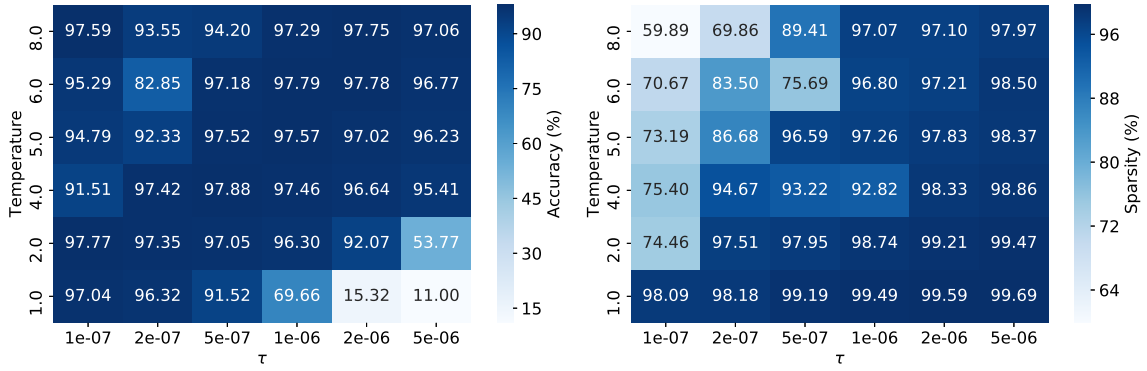
(a) Accuracy (b) Sparsity

Fig. 4.12 Grid Search for Inverse Gamma Prior Mean for Mixture Precision

τ and Temperature

τ and the temperature on the prediction targets were then tested together in a grid-search. A higher temperature leads to a smoother target distribution with less information contained, and a higher τ leads to greater sparsity at the cost of accuracy. Figure 4.13 shows a grid search over τ and temperature. The temperature on the prediction targets and τ can both be used to control the trade-off between accuracy and sparsity. High τ and temperature both lead to increased sparsity but this also coincides with low accuracy, as seen in the bottom right of the heatmap. At the opposite end, there is much higher accuracy but lower sparsity. Ideally, we would like to tread on the diagonal between the two to achieve high levels of compression and accuracy.

Overall, τ and temperature have not helped increase compression but moved results along the pareto front. The effects of τ and temperature are similar in that they help trade accuracy for sparsity.

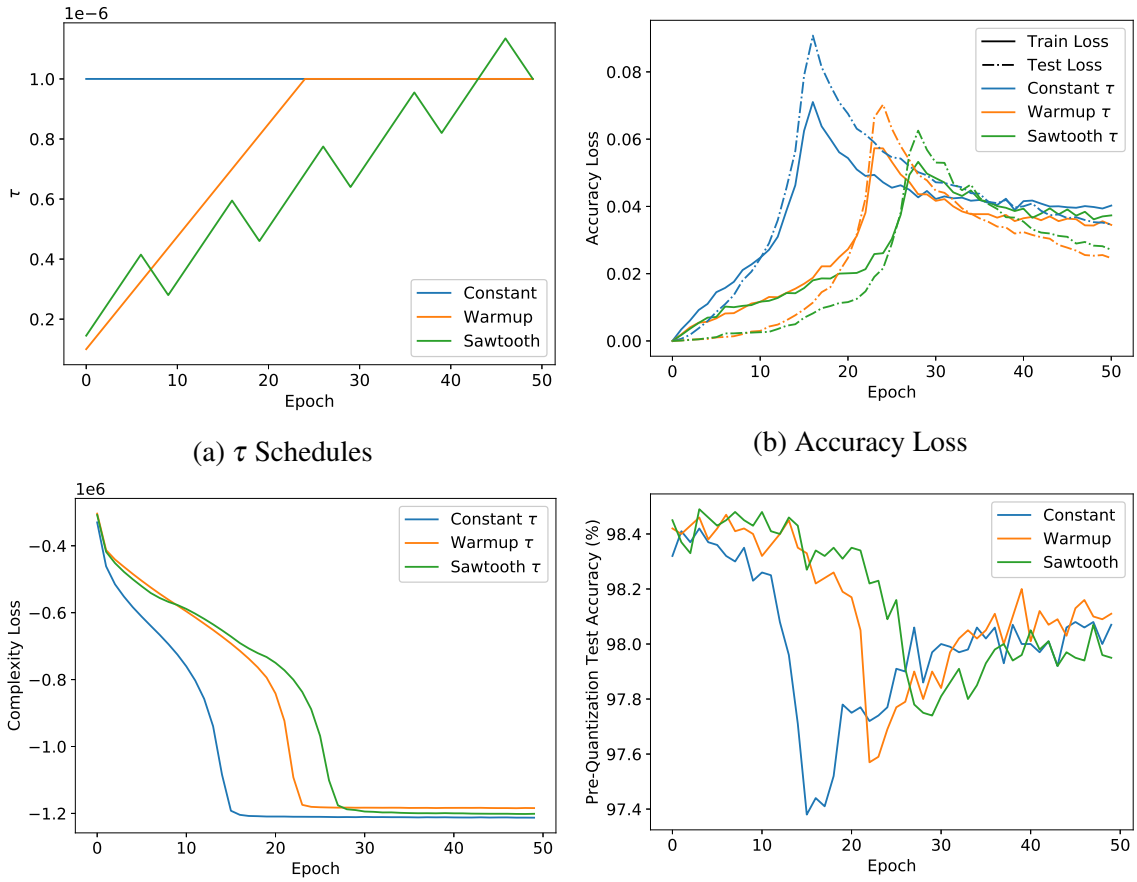


(a) Accuracy (b) Sparsity

Fig. 4.13 Temperature and τ Grid Search

The schedule for the trade-off parameter τ was also investigated. 3 different schemes were tried, as shown in figure 4.14 (a). The constant schedule stays at the maximum τ. The warmup pattern starts at a low value to decrease the accuracy drop during the clustering readjustment. The sawtooth pattern tries to find the optimal state for accuracy after each τ increase.

As we are using knowledge distillation, both test and training accuracy loss begin at zero. We see the accuracy loss increase and complexity loss decrease as the parameters readjust for clustering as before. Using a lower τ does decrease the accuracy drop at this stage, but regardless of the schedule, they converge to similar results. Table 4.7 shows slight differences between the different schedules, essentially placing them along different points of the pareto front. The use of low initial τ value also delays the clustering readjustment as can be seen in figure 4.14 (c). We use a constant τ for the remaining experiments.



(a) τ Schedules

(b) Accuracy Loss

(c) Complexity Loss

(d) Test Accuracy

NB: Before Quantisation and Pruning

Fig. 4.14 Effects of Different τ Schedules

τ Schedule	Accuracy (%)	Sparsity (%)
Constant	97.7	97.2
Warmup	97.8	96.5
Sawtooth	97.6	96.9

Table 4.7 Results for Different τ Schedules

Loss Functions

Once we start using the transfer set, we can use a choice of different loss functions. The original implementation of knowledge distillation by Caruana et. al [1] used the logits directly (see eq. 3.9). Later on, temperature controlled targets were used to soften the output

distribution and a cross-entropy loss function was used (see eq. 3.11) [9]. Another option is to use the mean-squared error loss on the temperature controlled predictions.

Figure 4.15 shows the accuracy and sparsity scatter plots for all 3 techniques with different complexity and accuracy trade-offs. If we were to draw a pareto front for each of the different loss functions, it is quite clear that using pure logits does not let the model exceed 93% accuracy with over 90% sparsity, making it a far worse performer than either cross-entropy or mean-squared error loss. Between mean-squared error and cross-entropy, the difference in performance is smaller with mean-squared error loss performing marginally better.

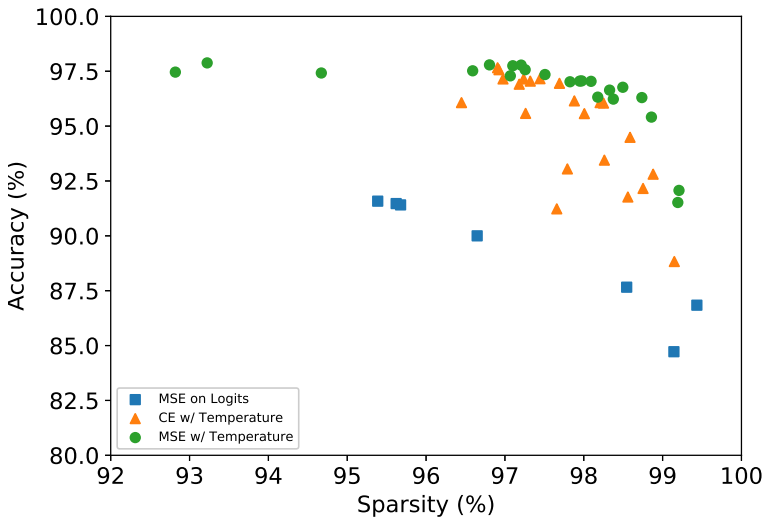


Fig. 4.15 Effects of Different Loss Functions on Soft-weight Sharing with Distillation

To build an intuition as to why these loss functions performed differently, we assume a simple distribution of logits where the maximum logit has a target value of 10. The value of the prediction for this logit is varied from 0 to 20. Figure 4.16 plots the errors and negative error gradients for different loss functions. The error has been normalised from 0 to 1 for all loss functions. Pure logits exhibit a gradient throughout, adding severe constraints to the algorithm and not allowing it to optimise alongside the complexity constraints. Using a softmax remedies this by reducing error gradients if the distance from target becomes too large.

Cross-entropy with temperature tries to increase the value beyond 10 whereas mean-squared error tries to keep the prediction at its correct value of 10. The largest prediction attaining a higher value should not cause accuracy problems, however, with a softened distribution many values are brought closer together and optimisation becomes more constrained if all targets try to increase beyond actual values.

The derivatives of the two loss function with respect to the logits (z_i) also explain this behaviour as MSE gradient is proportional to L1 distance between target and prediction. Cross-entropy gradient on the other hand divides true target by prediction, ensuring prediction to be somewhat larger than actual target. The loss functions are presented below with detailed derivation in appendix A.

$$\frac{\partial \mathscr{L}_{CE}}{\partial z_i} = -\sum_i y_i \frac{1}{\sigma_i} \times \frac{\partial \sigma_i}{\partial z_i} \tag{4.1}$$

$$\frac{\partial \mathscr{L}_{MSE}}{\partial z_i} = \frac{1}{N} \sum_{i=1}^{N} 2(\sigma_i - y_i) \times \frac{\partial \sigma_i}{\partial z_i} \tag{4.2}$$

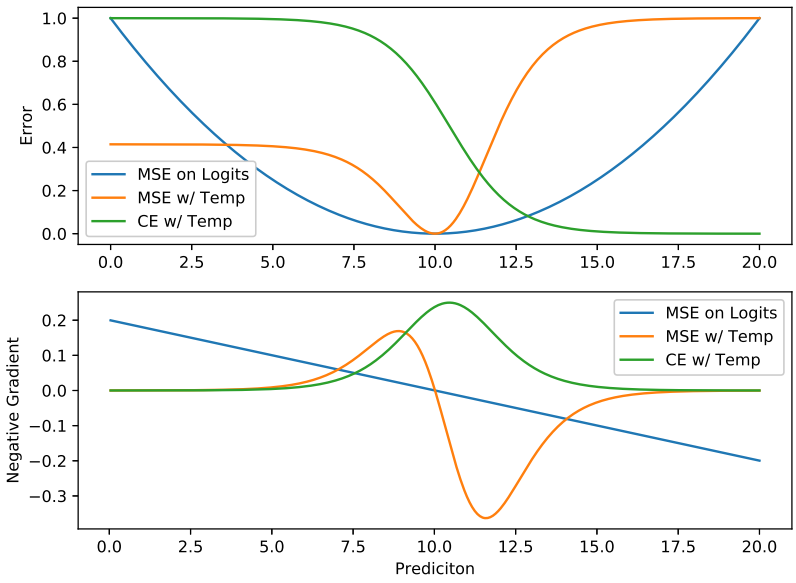


Fig. 4.16 Error and Gradients for Different Loss Functions

Mean-squared error is used from here onwards for distillation as it performs better.

Synthetic Data

One of the advantages of using knowledge distillation is that true labels are not required. If we have a sufficiently accurate input data generator or a large amount of unlabelled data, we can still carry out compression without access to the original dataset. We experiment with the use of synthetic data created from 1,000 images from the MNIST and FashionMNIST dataset. The mean and variance of each pixel is calculated over 1,000 images and Gaussian sampling is used to create 10,000 synthetic images.

Soft-weight sharing with and without distillation is carried out using the 1,000 digits. The synthetic data with 10,000 points is then used to carry out soft-weight sharing with distillation. Table 4.8 shows the results of this experiment, comparing all 3 procedures. Without the use of synthetic data, the algorithms perform poorly with larger drops of nearly 50% in accuracy. The synthetic data provides additional constraints that mitigate the accuracy loss, reducing it to roughly 15%, while still providing a high level of sparsity.

Method	MNIST		FashionMNIST	
	AC	SP	AC	SP
Soft-weight Sharing (SWS)	42.7	99.2	57.8	97.7
SWS with Knowledge Distillation (KD)	46.2	99.7	35.0	98.7
SWS with KD on Synthetic Data	84.7	99.5	73.8	98.3

Table 4.8 Use of Synthetic Data for Compression

**Key:* *AC: Compressed Test Accuracy (%), SP: Compressed Sparsity(%)*

Summary Results

The layer-by-layer breakdown for sparsity after incorporating distillation into soft-weight sharing is given in figure 4.9. The patterns are similar to pure soft-weight sharing, with the densest layer accounting for majority of the sparsity.

Layer	#[W]	(%)	#[W≠0]
FC1	235500	**98.13**	4414
FC2	30100	92.77	2175
FC3	1010	28.22	725
Total	266610	97.3	7314

Layer	#[W]	(%)	#[W≠0]
Conv1	650	34.62	425
Conv2	11300	80.99	2148
FC1	625500	**99.44**	3526
FC2	5010	75.37	1234
Total	642460	98.9	7333

Table 4.9 Layer-wise Sparsity on MNIST with Distillation

Left: LeNet-300-100. **Right:** Custom CNN

Further experiments were run on the FashionMNIST dataset and the custom CNN model. Incorporating knowledge distillation has not improved on the soft-weight sharing results and achieves roughly similar performance.

Model	MNIST				FashionMNIST			
	OAC	AC	SP	CR	OAC	AC	SP	CR
LeNet-300-100	98.3	97.6	97.3	81.8	89.6	84.4	98.1	103.2
Custom CNN	99.0	97.9	98.9	129.3	89.9	87.0	98.4	115.5

Table 4.10 Summary Results for Soft-weight Sharing Compression with Distillation

**Key:* *OAC: Original Test Accuracy (%), AC: Compressed Test Accuracy (%), SP: Compressed Sparsity(%), CR: Compression Rate*

Some additional results for distillation have been presented in appendix B.

4.3.3 Layer-wise Scaling

A proposed improvement to soft-weight sharing algorithm is allow each of the layers of the neural network to have a different scale for the mixture means. Table 4.11 shows the standard deviation of each layer after pre-training on MNIST, with the densest layer having the lowest spread. In soft-weight sharing, we found that the densest layer achieves the most sparsity and hence contributes to complexity loss to a greater extent. The remaining layers may not be able to achieve reasonable accuracy given the complexity constraints.

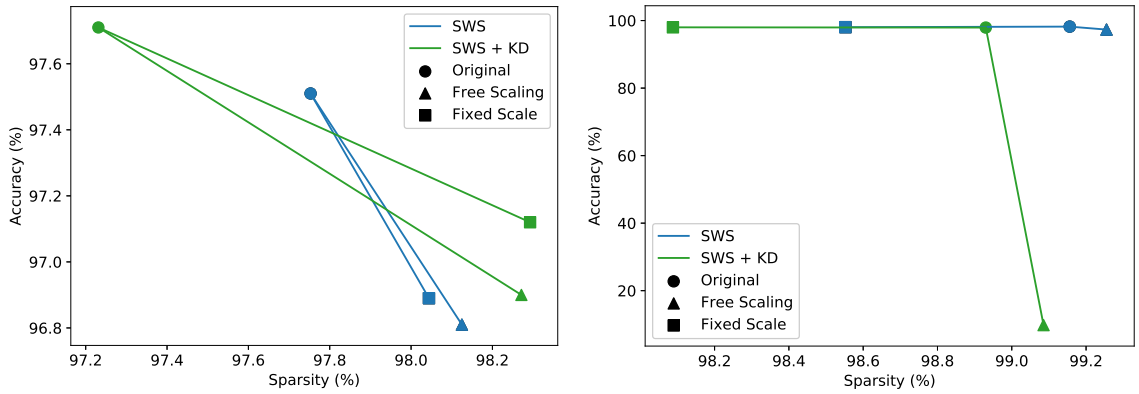
Layer	σ_L	$\frac{\sigma_L}{\sigma_{FC1}}$
FC1	0.15	1.00
FC2	0.17	1.13
FC3	0.23	1.47

Layer	σ_L	$\frac{\sigma_L}{\sigma_{Conv1}}$
Conv1	0.30	1.00
Conv2	0.23	0.78
FC1	0.16	0.52
FC2	0.20	0.67

Table 4.11 Absolute and Relative Standard Deviation in a Pre-trained MNIST Network

Left: LeNet-300-100. **Right:** Custom CNN

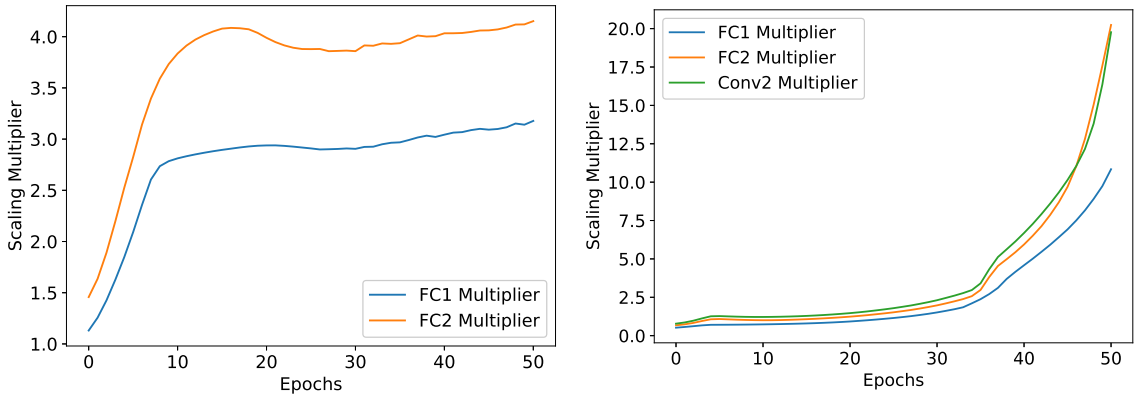
We allow for each of the layers to have a different scale. A scaling multiplier is introduced that divides the weight values during the calculation of complexity loss. Experiments are carried out with both fixed scaling multipliers and allowing the scales to be a free parameter. In the first case, we simply set the scaling multiplier as the relative standard deviation from the first layer, as in table 4.11. In the second case, we initialise the scale parameter as the relative standard deviation from the first layer. As the scaling is meant to be relative to the first layer, the first scaling multiplier is fixed to one with the remaining multipliers acting as free parameters.



(a) LeNet-300-100 (b) Custom CNN

Fig. 4.17 Soft-weight Sharing with Layer-wise Scaling on MNIST

The accuracy and sparsity scatter plot is shown in figure 4.17 for both fixed and variable scaling. With variable scaling multipliers, the sparsity increases but the accuracy has decreased considerably. In the case of Custom CNN, the free scaling multiplier breaks the classifier completely when used with knowledge distillation.

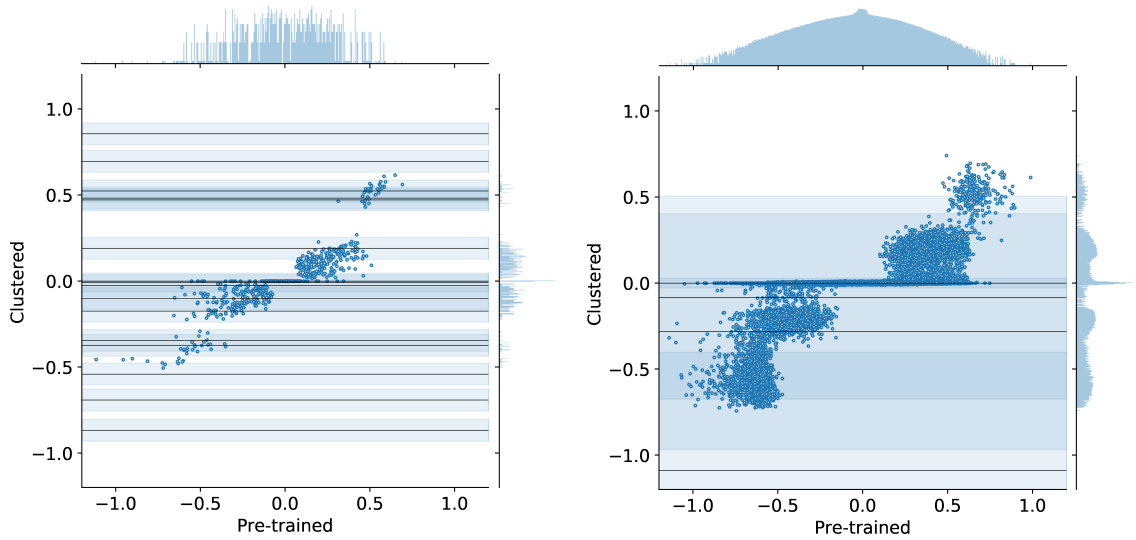


(a) LeNet-300-100 (b) Custom CNN

Fig. 4.18 Soft-weight Sharing with Layer-wise Scaling on MNIST

Using scaling as a free parameter has given disappointing results. To help understand why scaling does not work with soft-weight sharing, we plot the scales for each of the two networks but with high learning rate for the scale parameter in figure 4.18. The scales for all layers in both models rise quickly and keep an upward trend. The increased sparsity indicates that more parameters have been pruned to 0-value. This proves that zero-mean mixture with high mixing proportion ($\pi_j = 0.99$) can effectively attract parameters in other layers by increasing the scale as it squeezes all the weights or effectively widens the mixtures about 0. As more and more parameters are attracted to the zero-mean mixture due to the complexity loss, the unpruned accuracy remains high while the pruned accuracy falls drastically.

Figure 4.19 shows the first layer (Conv1) and third layer (FC1) of the Custom CNN which fails with scaling. The scale for the first layer is fixed to 1, hence there are many mixture means each point can be quantised to. The means deviate far out for FC1 with zero-mean mixture pruning away almost all the values. The pre-prune test accuracy 98.9% falls to 9.8% after pruning.



(a) Conv1

(b) FC1

NB: Remaining mixtures are outside -1<w_i<1

Fig. 4.19 Mixture Scaling in Custom CNN

The scaling has an effect on the precisions of the Gaussian mixtures which does not register in the accuracy loss. The spread of the mixtures increases with the scale. The accuracy drop only occurs once the pruning step is carried out. As pruning and quantisation are discontinuous processes, we cannot directly incorporate this error into our optimisation. A possible remedy for this effect is to scale the precisions of the Gaussian mixtures using the scaling multiplier, ensuring when a weight is scaled to a lower value, the spread of the zero-mean mixture is also effectively reduced. Hence, avoiding the pruning of critical parameters.

The fixed scaling multiplier is slightly more accurate and less sparse in the Custom CNN and slightly more sparse at the expense of accuracy for LeNet-300-100. The topology of the two networks can explain the results. Our implementation defines scaling relative from the initial layer. In the LeNet-300-100, the initial layer is the densest with the smallest spread making the relative scaling >1. In the Custom CNN, the initial layer has the largest spread, making the relative scaling <1. Dividing the weights by <1 effectively allows the parameters in the layer to quantise to mixtures means further away from zero, increasing accuracy but decreasing sparsity.

The effect is more pronounced for the distilled version of soft-weight sharing. The soft targets allow more flexibility during optimisation due to the smoother distribution. Soft-

weight sharing's harder constraints can help mitigate accuracy drops.

Layer	#[W]	(%)	#[W≠0]
FC1	235500	**98.40**	3771
FC2	30100	93.36	1999
FC3	1010	34.16	665
Total	266610	97.6	6435

Layer	#[W]	(%)	#[W≠0]
Conv1	650	52.92	306
Conv2	11300	89.25	1215
FC1	625500	**98.85**	7176
FC2	5010	87.94	604
Total	642460	98.6	9301

Table 4.12 Layer-wise Sparsity on with Soft-weight Sharing with Fixed Scaling on MNIST.

Left: LeNet-300-100. **Right:** Custom CNN

Table 4.12 shows the sparsity broken down by each layer for soft-weight sharing with a fixed scale multiplier. Incorporating a fixed scale multiplier, we hoped to increase sparsity in the layers with greatest standard deviation. In FC3 for LeNet-300-100 and Conv1 for the Custom CNN, the sparsity is still quite low.

Figure 4.20 shows the results of a fixed scaling multiplier with soft-weight sharing and distillation modification. Both perform reasonably well, roughly at similar performance as vanilla soft-weight sharing.

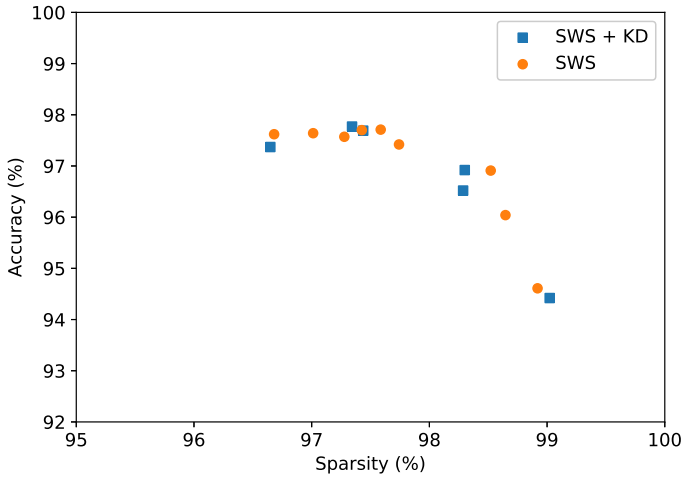


Fig. 4.20 Accuracy and Sparsity for Soft-weight Sharing with Layer-wise Scaling on MNIST

Table 4.13 presents the results of the fixed scaling multiplier. The models still achieve above 80× compression with less than 1% accuracy drop. Both methods perform roughly similarly on LeNet-300-100 while Custom CNN compresses better with pure soft-weight sharing.

Model	SWS				SWS and Distillation			
	OAC	AC	SP	CR	OAC	AC	SP	CR
LeNet-300-100	98.3	97.7	97.6	89.8	98.3	97.7	97.4	85.7
Custom CNN	99.0	98.0	98.6	124.3	99.0	98.0	98.1	103.8

Table 4.13 Summary Results for Soft-weight Sharing with Fixed Scale Multiplier on MNIST

**Key:* *OAC: Original Test Accuracy (%), AC: Compressed Test Accuracy (%), SP: Compressed Sparsity(%), CR: Compression Rate*

Table 4.14 shows similar testing carried out on the FashionMNIST dataset. The LeNet-300-100 has a fall in accuracy but compresses well. Between the original and distilled soft-weight sharing there is a trade-off between sparsity and accuracy. The remaining models compress well at over 80× compression rate.

Model	SWS				SWS and Distillation			
	OAC	AC	SP	CR	OAC	AC	SP	CR
LeNet-300-100	89.6	85.4	97.4	86.1	89.6	83.2	98.2	106.4
Custom CNN	89.9	88.0	97.3	89.5	89.9	88.5	97.1	80.4

Table 4.14 Summary Results for Soft-weight Sharing with Fixed Scale Multiplier on FashionMNIST

**Key:* *OAC: Original Test Accuracy (%), AC: Compressed Test Accuracy (%), SP: Compressed Sparsity(%), CR: Compression Rate*

4.3.4 Layer-wise Compression

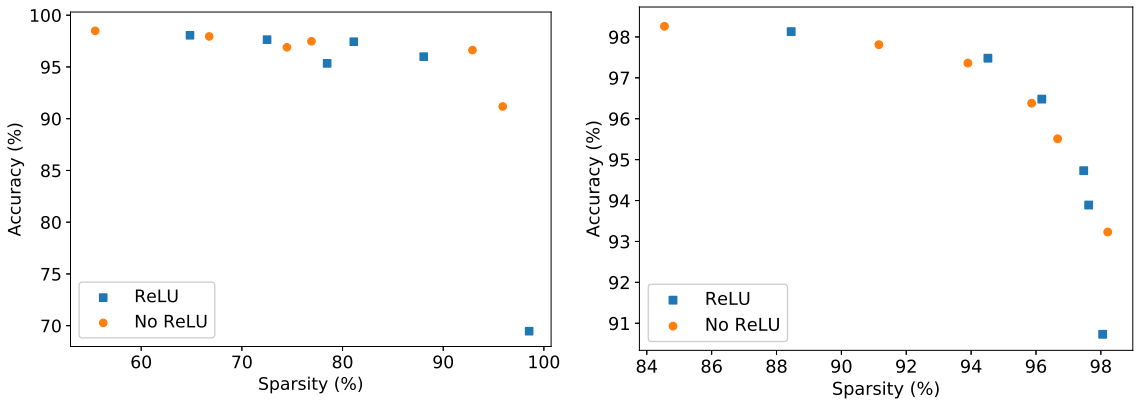
Layer-wise compression is carried out to test whether better optimization can be performed with soft-weight sharing when the objective is to mimic each layer. For knowledge distillation, we extracted a transfer set of predictions for the targets. Similarly, for layer-wise compression, we can extract pre-trained network's predictions at each hidden layer. To test the hypothesis successfully, we need to match the clustered activations to the pre-trained

activations at each hidden layer. We have proposed two methods for this.

Method 1: Independent Compression

In the first method, all the layers are trained completely independent of each other. Each layer will has a separate GMM prior and error is not backpropagated between the different layers.

Since compression is carried out independent of other layers, we can examine the effect of compression in a particular layer by taking a fully trained network and replacing the original layer with the compressed layer. Our initial experiments checks whether the use of ReLU activation in hidden layer loss calculation causes any problems. Figure 4.21 shows that the performance is similar with or without the use of activation. We continue without the use of activation in remaining experiments.



(a) Layer 1

(b) Layer 2

Fig. 4.21 Effect of Activations for Layer-wise Compression

A major source of error for this method is that no error is backpropagated from the targets, hence each layer can introduce errors that may not be corrected. On the other hand, as all the layers are compressed independently, different hyperparameters can be assigned throughout. We use different trade-off parameters to find the sparsest compressed layer model for a given accuracy drop (ε).

Table 4.15 shows the sparsest layer found in our search for 1%, 2%, 3%, and 10% accuracy fall per layer. Although FC3 compresses much better at 81% against soft-weight sharing's 39%, and FC2 performs at nearly a similar level, the method is unable to compress

FC1 to the degree needed for this modification to be advantageous. FC1 is much denser and accounts for over 88% of the weights in the network, hence dominating the compression. Yet, even with a 6% drop in accuracy, it only attains 95.9% sparsity compared to 98.5% in the original algorithm. The first layer alone effectively caps the amount of compression that can be carried out.

Furthermore, we see that even if $\varepsilon \leq 1$ for all 3 layers, the error compounds significantly. Despite each layer holding less than 1% accuracy drop from a fully-trained network, the overall accuracy drop when combined is 7%. This is unfeasible, especially for deep neural networks with hundreds of layers.

One possible method to remedy this is to rejoin and retrain the network for a few epochs to fix these misalignment errors. However, this defeats the purpose of carrying out compression of each layer independently and potentially in parallel. Furthermore, the reason why FC3 can achieve a high level for sparsity and accuracy here is that normally the densest layer dominates the prior. By assigning each layer different hyperparameters and priors, greater sparsity has been acheived. Unifying the network and sharing hyperparameters and priors will not only significantly change the final optimised state, requiring more epochs for clustering but also would also lose the benefits of separately tuned and trained compressed layers.

Another possible but untested remedy is to include random noise at the input of each layer while keeping the same targets. This is somewhat analogous to the encoder in the de-noising autoencoder, with the latent representation replaced by the targets. The neural network should become more robust to the minute changes between the layers, decreasing the accuracy drop when connecting the layers. However, since the densest FC1 layer is not able to compress to the same extent as soft-weight sharing in the first place, any methods to reduce accuracy loss at the unification step will still not help achieve state of the art compression.

ε	Layer	Accuracy (%)	Sparsity (%)
1.0	FC1	97.48	76.91
	FC2	97.36	93.90
	FC3	97.46	81.49
	Overall	**91.27**	78.41
2.0	FC1	96.63	92.91
	FC2	96.38	95.87
	FC3	97.46	81.49
	Overall	84.28	92.73
3.0	FC1	96.63	92.91
	FC2	95.51	96.67
	FC3	97.46	81.49
	Overall	85.56	92.88
10.0	FC1	91.18	**95.91**
	FC2	93.23	98.21
	FC3	97.46	81.49
	Overall	74.80	**95.96**

Table 4.15 Results of Method 1 for Layer-wise Compression

Method 2: Layer-wise Loss Accumulation

In the second method, we train the whole network together but incorporate a loss at each layer (see eq. 3.12). The GMM prior is shared over the whole network and hence the number of quantised values required for the codebook are simply the number of mixtures. The error is also backpropagated, ensuring the whole network is perfectly aligned.

Figure 4.22 shows the scatter plot of accuracy against sparsity. Method 2 clearly outperforms method 1. A higher accuracy region of $> 95\%$ and $> 95\%$ sparsity is present which was not the case with the first method which was unable to attain an accuracy greater than 91.3%. This compression method is severely sensitive to the trade-off parameter. The pareto front travels from 96% accuracy at 72% sparsity to 98% sparsity at roughly the same accuracy.

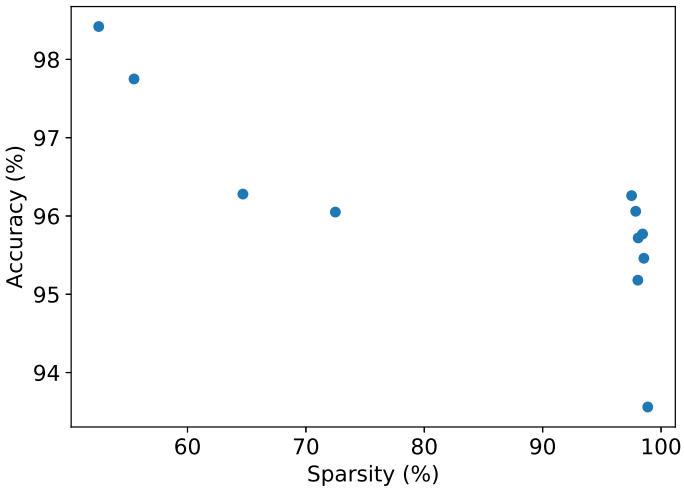


Fig. 4.22 Soft-weight Sharing with Layer-wise Compression (Method 2) on MNIST

Table 4.16 shows the layer-wise sparsity when accuracy of over 96% is required. We can only attain a sparsity of 65% in LeNet-300-100 but the Custom CNN fairs better. The constraints from the activations of FC1 did not allow sufficient compression as FC1 could only attain 63% sparsity whereas it normally attains 90%+.

Layer	#[W]	(%)	#[W≠0]
FC1	235500	**62.66**	87947
FC2	30100	**83.22**	5052
FC3	1010	33.56	671
Total	266610	64.9	93670

Layer	#[W]	(%)	#[W≠0]
Conv1	650	30.31	453
Conv2	11300	80.63	2189
FC1	625500	**99.16**	5227
FC2	5010	27.78	3618
Total	642460	98.2	11487

Table 4.16 Layer-wise Sparsity for Layer-wise Compression on MNIST.

Left: LeNet-300-100. **Right:** Custom CNN

Table 4.17 shows results on different datasets and models. The LeNet-300-100 fails to compress sufficiently while trying to maintain high accuracy. The Custom CNN still achieves over 100× compression, but this is still worse off than previous methods. The added constraints which were meant to help optimise the model instead only hindered compression.

Model	MNIST				FashionMNIST			
	OAC	AC	SP	CR	OAC	AC	SP	CR
LeNet-300-100	98.3	97.6	55.4	7.2	89.6	79.7	54.2	7.0
Custom CNN	99.0	97.0	98.2	105.6	89.9	82.1	98.4	108.5

Table 4.17 Summary Results for Layer-wise Compression

Key: *OAC: Original Test Accuracy (%), AC: Compressed Test Accuracy (%), SP: Compressed Sparsity(%), CR: Compression Rate*

This chapter has presented detailed results and analysis which will help us answer our research questions in the next chapter.

Chapter 5

Discussion

We investigated 3 research questions posed earlier in the thesis. The results from experiments carried out in chapter 4 can help us answer these questions sufficiently. Summary results are presented in the form of a plot for accuracy and sparsity on LeNet-300-100 in figure 5.1. Table 5.1 gives the summary results across different datasets and models with the compression rate and accuracy change. The results for variable layer scaling and independent layer compression (method 1) are not listed as these methods were not able to maintain accuracy. Fixed scaling, and layer-wise loss accumulation (method 2) results are present.

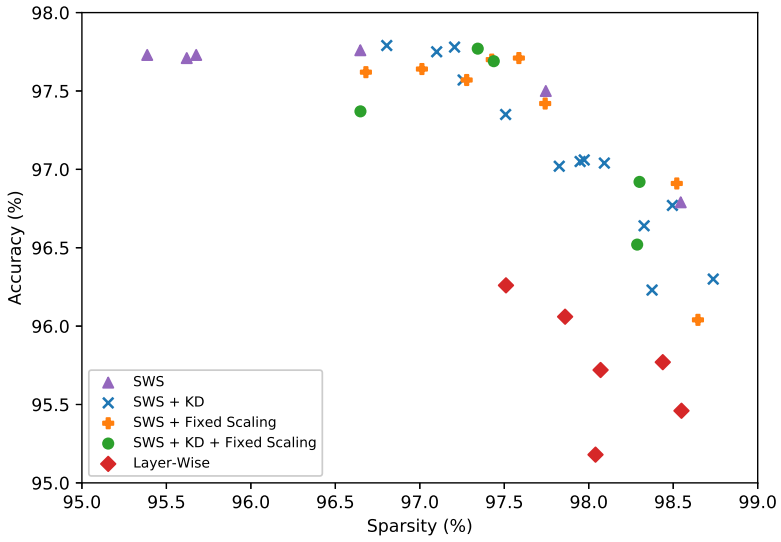


Fig. 5.1 Accuracy and Sparsity Plot for all Methods

Method	LeNet-300-100				Custom CNN			
	MNIST		FashionMNIST		MNIST		FashionMNIST	
	ΔACC	CR	ΔACC	CR	ΔACC	CR	ΔACC	CR
SWS	0.8	92.3	3.8	86.3	0.8	183.7	1.4	150.6
SWS + KD	0.7	81.8	5.2	103.2	1.1	129.3	2.9	115.5
SWS + Fixed Scaling	0.6	89.8	4.2	86.1	1	124.3	1.9	89.5
SWS + KD + Fixed Scaling	0.6	85.7	6.4	106.4	1	103.8	1.4	80.4
Layerwise	0.7	7.2	9.9	7	2	105.6	7.8	108.5

Table 5.1 Summary Results for Experiments

Top: LeNet-300-100 **Bottom:** Custom CNN

**Key:* *ΔACC: Accuracy Drop (%), CR: Compression Rate, SWS: Soft-weight Sharing, KD: Knowledge Distillation*

5.1 Effects of Knowledge Distillation

The first research question examines the effects of distillation on soft-weight sharing. Attempting distillation with pure logits added unnecessary constraints, degrading performance. However, the use of mean-squared error with temperature controlled logits gives nearly similar results to soft-weight sharing. The pareto front for soft-weight sharing, with and without distillation in figure 5.1 is seen to be quite close.

The use of distillation adds another hyperparameter, temperature, which smoothes the target distribution. The effect of this is essentially similar to τ, trading compression for accuracy. The temperature moves around the pareto front but does not give significantly better results. However, with the smoother distribution, the method becomes less robust. Tuning of inverse Gamma prior on the mixture precisions has shown greater stochasticity with knowledge distillation than without. Similarly, table 5.1 shows that distillation performs worse on the Custom CNN than soft-weight sharing while showing comparable performance in LeNet-300-100. This could be due to insufficient hyperparameter tuning.

The one area where distillation has shown to be significantly helpful is the use of synthetic data without labels. Experiments have shown that with only 1,000 points the accuracies of soft-weight sharing drop by 50%, but creating 10,000 synthetic data points and using distillation stops the accuracy drop around 15%. A better data augmentation method that

Gaussian sampling with more data points should perform even better.

Overall, knowledge distillation performs are roughly similar levels to soft-weight sharing with higher sensitivity to hyperparameters but no need for true labels on the training set.

5.2 Effects of Incorporating Layer-wise Scaling

The second research questions asks whether incorporating scaling into soft-weight sharing can help improve compression. A variable scaling method and a fixed scaling method were tried. The variable scaling method did not work with soft-weight sharing as it encourages scalable layer weights to simply collapse to the zero-mean mixture due to its high mixing proportion. At the pruning stage, there is a significant drop in accuracy as most of the scalable layers are pruned.

Using fixed scaling by simply assuming the relative standard deviation from the pre-trained networks distribution does not allow the collapse to zero-mean. However, there little to no improvement on the actual results with our without distillation. In either case, the performance simply moves along the pareto front, either increasing accuracy at the cost of compression or vice versa.

In current experimental setup, scaling does not improve soft-weight sharing. Methods to overcome the collapse of scalable layer to zero-mean mixture should be investigated. Independent layer-wise compression shows that the FC3 in LeNet-300-100 could achieve 81% sparsity with less than 1% drop in accuracy when trained with an independent prior. Hence, it is likely that these layers can be further compressed if these issues are overcome.

5.3 Is Layer-wise Compression Better?

The third research question looks into whether better compression can be acheived if we compress layer-wise and mimic each layer in the pre-trained network. Carrying out such compression completely independently gave disappointing results. We could not attain higher sparsity than 93% without significantly degrading accuracy in the densest layer. This shows that the constraints of layer-wise compression simply do not encourage sparsity. Although the model simply translates to a regression problem, matching the hidden layer activations

reduce flexibility by far too much.

Further testing was done by using a loss function that trained the network as a whole, but also accounted for the hidden unit activations. This method improved compression over the previous method while mimicking each layer. However, it is extremely sensitive to the trade-off hyperparameter and did not improve upon the original soft-weight sharing. Figure 5.1 shows layer-wise compression further away from the pareto front. The results on LeNet-300-100 for also show the compression rate to be merely 7 for less than 1% accuracy drop on MNIST and 10% accuracy drop on FashionMNIST.

Both methods verify that layer-wise compression degrades performance of soft-weight sharing as it adds unnecessary constraints on the network, reducing sparsity.

All 3 research questions proposed improvements to soft-weight sharing which seemed viable initially. Through experimentation, we have found that soft-weight sharing has performed remarkably well and the modifications did not yield the benefits we expected. The results, while not positive, are informative.

Chapter 6

Conclusion

The thesis investigated neural network compression by attempting to improve a near state-of-the-art algorithm. All in all, we found that the original soft-weight sharing performs quite well and the suggested modifications could not improve upon it for various reasons.

Knowledge distillation can still be somewhat useful in some applications as it does not require training data labels and can also make use of synthetic data. Variable scaling should be further investigated with methods that prevent layers from collapsing to zero-mean mixture. The results showed that carrying out compression layer-wise does not help overall compression but also showed that the less dense layers had capacity to be further compressed.

Although experiments and tests were run across different models and datasets, the limitation on computing resources mean that we could not actually try our methods on an actual deep neural network. An epoch of clustering takes roughly 1 minute on an Nvidia K80 with LeNet-300-100, making deeper models with millions of parameters unfeasible. It would be interesting to see how deeper and more parametrised networks perform with these algorithms as these networks are where compression will likely be used.

Many new avenues and methods for compression are being introduced. Currently, variational dropout methods achieve state-of-the-art compression, with soft-weight sharing not far behind. Methods which look for regularity in pruning provide more pratical benefits and could be incorporated with soft-weight sharing. Results and intuition from this thesis can help inform research directions for various methods as neural network compression is likely to remain a topic of interest for some time.

References

[1] Ba, J. and Caruana, R. (2014). Do deep nets really need to be deep? In *Advances in neural information processing systems*, pages 2654–2662.

[2] Bawden, T. (2016). Global warming: Data centres to consume three times as much energy in next decade, experts warn. https://www.independent.co.uk/environment/global-warming-data-centres-to-consume-three-times-as-much-energy-in-next-decade-experts-warn-a68 html, Accessed: 04-08-2018.

[3] Canziani, A., Paszke, A., and Culurciello, E. (2016). An analysis of deep neural network models for practical applications. *arXiv preprint arXiv:1605.07678*.

[4] Frankle, J. and Carbin, M. (2018). The lottery ticket hypothesis: Training pruned neural networks. *arXiv preprint arXiv:1803.03635*.

[5] Gers, F. A., Schmidhuber, J., and Cummins, F. (1999). Learning to forget: Continual prediction with lstm.

[6] Han, S., Mao, H., and Dally, W. J. (2015a). Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding. *arXiv preprint arXiv:1510.00149*.

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

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

[9] Hinton, G., Vinyals, O., and Dean, J. (2015). Distilling the knowledge in a neural network. *arXiv preprint arXiv:1503.02531*.

[10] Krizhevsky, A., Sutskever, I., and Hinton, G. E. (2012). Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems*, pages 1097–1105.

[11] LeCun, Y., Bottou, L., Bengio, Y., and Haffner, P. (1998). Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324.

[12] Li, C., Farkhoor, H., Liu, R., and Yosinski, J. (2018). Measuring the intrinsic dimension of objective landscapes. *arXiv preprint arXiv:1804.08838*.

[13] Louizos, C., Ullrich, K., and Welling, M. (2017). Bayesian compression for deep learning. In *Advances in Neural Information Processing Systems*, pages 3288–3298.

[14] Mao, H., Han, S., Pool, J., Li, W., Liu, X., Wang, Y., and Dally, W. J. (2017). Exploring the regularity of sparse structure in convolutional neural networks. *arXiv preprint arXiv:1705.08922*.

[15] Molchanov, D., Ashukha, A., and Vetrov, D. (2017). Variational dropout sparsifies deep neural networks. *arXiv preprint arXiv:1701.05369*.

[16] Nowlan, S. J. and Hinton, G. E. (1992). Simplifying neural networks by soft weight-sharing. *Neural computation*, 4(4):473–493.

[17] Rastegari, M., Ordonez, V., Redmon, J., and Farhadi, A. (2016). Xnor-net: Imagenet classification using binary convolutional neural networks. In *European Conference on Computer Vision*, pages 525–542. Springer.

[18] Ravi, S. (2017). Projectionnet: Learning efficient on-device deep networks using neural projections. *arXiv preprint arXiv:1708.00630*.

[19] Simonyan, K. and Zisserman, A. (2014). Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556*.

[20] Srivastava, N., Hinton, G., Krizhevsky, A., Sutskever, I., and Salakhutdinov, R. (2014). Dropout: a simple way to prevent neural networks from overfitting. *The Journal of Machine Learning Research*, 15(1):1929–1958.

[21] Szegedy, C., Liu, W., Jia, Y., Sermanet, P., Reed, S., Anguelov, D., Erhan, D., Vanhoucke, V., and Rabinovich, A. (2015). Going deeper with convolutions. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 1–9.

[22] TensorFlow (2018). Fixed point quantization. https://www.tensorflow.org/performance/quantization, Accessed: 17-07-2018.

[23] Ullrich, K. (2017). A tutorial on 'soft weight-sharing for neural network compression' published at ICLR2017. https://github.com/KarenUllrich/Tutorial-SoftWeightSharingForNNCompression.

[24] Ullrich, K., Meeds, E., and Welling, M. (2017). Soft weight-sharing for neural network compression. *arXiv preprint arXiv:1702.04008*.

[25] Walsh, C. and Huttenlocher, P. (2013). 502(7470):172–172. *Nature*.

[26] Wen, W., Wu, C., Wang, Y., Chen, Y., and Li, H. (2016). Learning structured sparsity in deep neural networks. In *Advances in Neural Information Processing Systems*, pages 2074–2082.

[27] Zeiler, M. D. and Fergus, R. (2014). Visualizing and understanding convolutional networks. In *European conference on computer vision*, pages 818–833. Springer.

Appendix A

Mathematical Derivations

A.1 Backpropogation Derivatives

Derivative of Softmax with Temperature

Approximation at high temperature:

$$\begin{aligned} \sigma_i &= \frac{\exp(z_i/T)}{\sum_k \exp(z_k/T)} \approx \frac{1+(z_i)/T}{N+\sum_k z_k/T} \\ \sigma_i\sigma_j &= \frac{\exp((z_i+z_j)/T)}{(\sum_k \exp(z_k/T))^2} \approx \frac{1+(z_i+z_j)/T}{(N+\sum_k z_k/T)^2} \end{aligned} \tag{A.1}$$

Approximation assuming zero-mean logits ($\sum_k z_k/T = 0$):

$$\begin{aligned} \sigma_i &\approx \frac{1+(z_i)/T}{N+\sum_k z_k/T} \approx \frac{1+z_i}{NT} \\ \sigma_i\sigma_j &\approx \frac{1+(z_i+z_j)/T}{(N+\sum_k z_k/T)^2} \approx \frac{1+z_i+z_j}{N^2T} \end{aligned} \tag{A.2}$$

Softmax derivative if $i = j$:

$$
\begin{aligned}
\sigma_i &= \frac{\exp(z_i/T)}{\sum_k \exp(z_k/T)} \\
\frac{\partial \sigma_i}{\partial z_j} &= \frac{\frac{1}{T}\exp(z_i/T)\sum_k \exp(z_k/T) - \frac{1}{T}\big(\exp(z_i/T)\big)^2}{\big(\sum_k \exp(z_k/T)\big)^2} \\
\frac{\partial \sigma_i}{\partial z_j} &= \frac{\frac{1}{T}\exp(z_i/T)\big[\sum_k \exp(z_k/T) - \exp(z_i/T)\big]}{\big(\sum_k \exp(z_k/T)\big)^2} \\
\frac{\partial \sigma_i}{\partial z_j} &= \frac{1}{T}\frac{\exp(z_i/T)}{\sum_k \exp(z_k/T)}\frac{\big[\sum_k \exp(z_k/T) - \exp(z_i/T)\big]}{\sum_k \exp(z_k/T)} \\
\frac{\partial \sigma_i}{\partial z_j} &= \frac{1}{T}\sigma_i(1-\sigma_i) \\
\frac{\partial \sigma_i}{\partial z_j} &\approx \frac{1+z_i}{NT^2} - \frac{1+2z_i}{N^2T^2}
\end{aligned}
\tag{A.3}
$$

Softmax derivative if $i \neq j$:

$$
\begin{aligned}
\sigma_i &= \frac{\exp(z_i/T)}{\sum_k \exp(z_k/T)} \\
\frac{\partial \sigma_i}{\partial z_j} &= \frac{0 - \frac{1}{T}\big[\exp(z_i/T)\exp(z_j/T)\big]}{\big(\sum_k \exp(z_k/T)\big)^2} \\
\frac{\partial \sigma_i}{\partial z_j} &= -\frac{1}{T}\frac{\exp(z_i/T)}{\sum_k \exp(z_k/T)}\frac{\exp(z_j/T)}{\sum_k \exp(z_k/T)} \\
\frac{\partial \sigma_i}{\partial z_j} &= -\frac{1}{T}\sigma_i\sigma_j \\
\frac{\partial \sigma_i}{\partial z_j} &\approx -\frac{1+z_i+z_j}{N^2T^2}
\end{aligned}
\tag{A.4}
$$

Hence proving, error gradient $\propto 1/T^2$

Derivative of Cross-Entropy Error

$$\begin{aligned}
\mathscr{L}_{CE} &= -\sum_i y_i \log(\sigma_i) \\
\frac{\partial \mathscr{L}_{CE}}{\partial z_i} &= -\sum_i y_i \frac{\partial \log(\sigma_i)}{\partial z_i} \\
\frac{\partial \mathscr{L}_{CE}}{\partial z_i} &= -\sum_i y_i \frac{\partial \log(\sigma_i)}{\partial \sigma_i} \times \frac{\partial \sigma_i}{\partial z_i} \\
\frac{\partial \mathscr{L}_{CE}}{\partial z_i} &= -\sum_i y_i \frac{1}{\sigma_i} \times \frac{\partial \sigma_i}{\partial z_i}
\end{aligned} \tag{A.5}$$

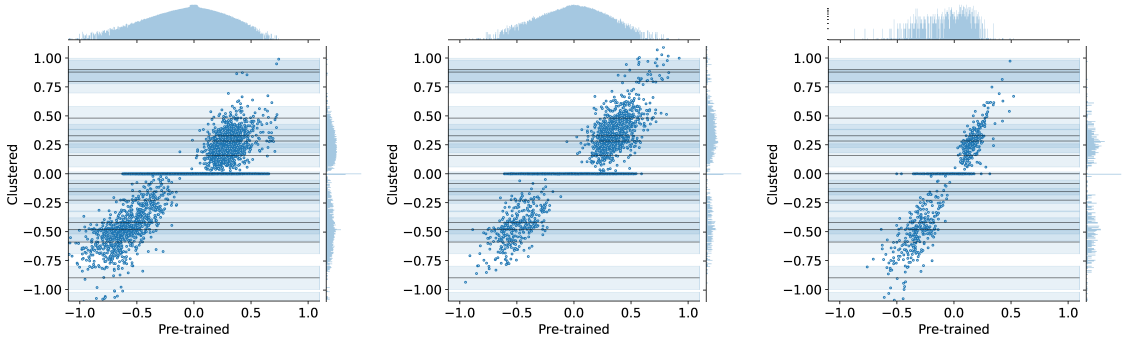
Derivative of Mean-Squared Error

$$\begin{aligned}
\mathscr{L}_{MSE} &= \frac{1}{N} \sum_{i=1}^{N} (\sigma_i - y_i)^2 \\
\frac{\partial \mathscr{L}_{MSE}}{\partial z_i} &= \frac{1}{N} \sum_{i=1}^{N} 2(\sigma_i - y_i) \times \frac{\partial \sigma_i}{\partial z_i}
\end{aligned} \tag{A.6}$$

Appendix B

Additional Results

Soft-weight Sharing



(a) FC1 (b) FC2 (c) FC3

Fig. B.1 Layer-wise Clustering in LeNet-300-100

Soft-weight Sharing with Distillation

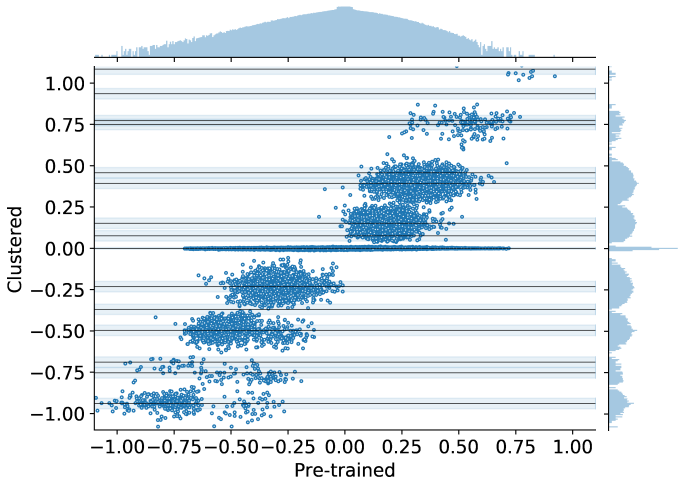


Fig. B.2 Clustering with Distillation in LeNet-300-100

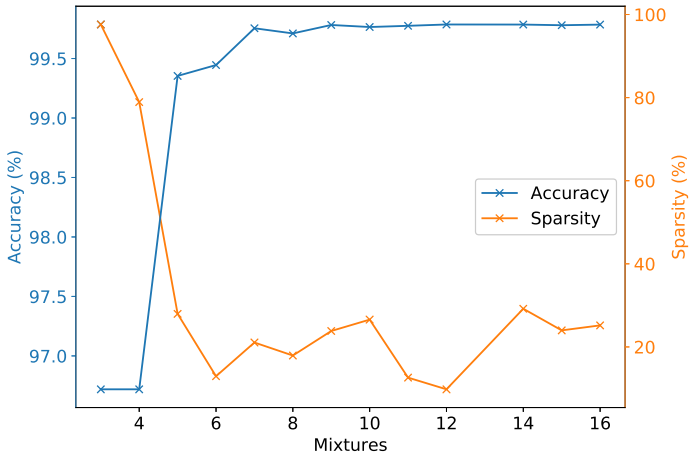
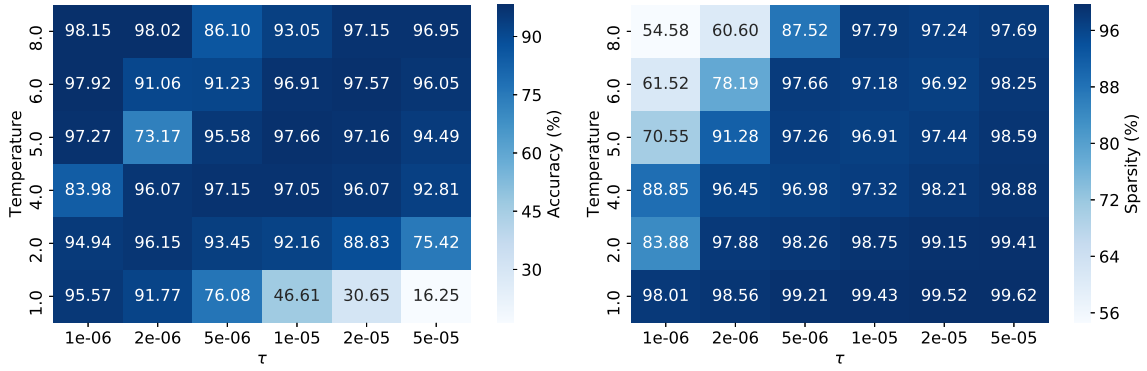


Fig. B.3 Effect of Number of Mixtures in Soft-weight Sharing with Distillation



(a) Accuracy (b) Sparsity

Fig. B.4 Temperature and τ Grid Search with Cross-Entropy Loss in Soft-weight Sharing with Distillation