THE UNIVERSITY
of EDINBURGH

This thesis has been submitted in fulfilment of the requirements for a postgraduate degree (e.g. PhD, MPhil, DClinPsychol) at the University of Edinburgh. Please note the following terms and conditions of use:

This work is protected by copyright and other intellectual property rights, which are retained by the thesis author, unless otherwise stated.
A copy can be downloaded for personal non-commercial research or study, without prior permission or charge.
This thesis cannot be reproduced or quoted extensively from without first obtaining permission in writing from the author.
The content must not be changed in any way or sold commercially in any format or medium without the formal permission of the author.
When referring to this work, full bibliographic details including the author, title, awarding institution and date of the thesis must be given.

TEST TIME COST SENSITIVITY IN MACHINE LEARNING

GAVIN GRAY

Doctor of Philosophy
School of Informatics
University of Edinburgh

2019

Gavin Gray:
Test Time Cost Sensitivity in Machine Learning
Doctor of Philosophy, 2019

supervisors:
Amos Storkey
D. K. Arvind

ABSTRACT

The use of deep neural networks has enabled machines to classify images, translate between languages and compete with humans in games. These achievements have been enabled by the large and expensive computational resources that are now available for training and running such networks. However, such a computational burden is highly undesirable in some settings. In this thesis we demonstrate how the computational expense of a machine learning algorithm may be reduced. This is possible because, until recently, most research in deep learning has focused on achieving better statistical results on benchmarks, rather than targeting efficiency. However, the learning process is flexible enough for us to control for the test-time computational expense that will be paid when the model is run in an application. To achieve this test-time computation sensitivity, a budget can be incorporated as part of the model. This budget expresses what costs we are willing to incur when we allocate resources at test time. Alternatively we can prescribe the size or computational resources we expect and use that to decide on the appropriate classification model. In either case, considering the resources available when building the model allows us to use it more effectively. In this thesis, we demonstrate methods to reduce the stored size, or floating point operations, of state-of-the-art classification models by an order of magnitude with little effect on their performance. Finally, we find that such compression can even be performed by simply changing the parameterisation of linear transforms used in the network. These results indicate that the design of learning systems can benefit from taking resource efficiency into account.

ACKNOWLEDGEMENTS

This thesis could not exist without the contribution of everyone around me, and I'm going to try and write down some of those people here.

Firstly, the Doctoral Training Centre for Neuroinformatics was a huge group of the most interesting people I've ever met. Without that community, I would not have been excited about research; not without talking to enthused PhD students in subjects ranging from insect robotics to incompleteness theorems. It brought me friendships that are worth far more to me than this thesis.

Academically, I must thank Elliot Crowley for taking interest in every idea I've thrown at him for the last few years. Joe Mellor and Matt Graham deserve mention for helping me solve various problems over the years (and Matt for developing this thesis template). Amos Storkey and D.K Arvind already got mentioned at the start of the thesis but they really were very important, so I'm mentioning them again here.

Personally, I must acknowledge my parents already have ultimate responsibility for this thesis. But, their continued support over the last two years in particular has been very important to me. I'd like to thank my mother for taking me outside when I would otherwise be stuck in a redrafting haze.

Finally, I would like to thank Alina Selega, whose support, if we could measure it, would convince you to award her the PhD instead of me. Luckily, it remains immeasurable.

DECLARATION

I declare that this thesis was composed by myself, that the work contained herein is my own except where explicitly stated otherwise in the text, and that this work has not been submitted for any other degree or professional qualification except as specified.

Edinburgh, 2019

Gavin Gray, October 10, 2019

CONTENTS

Abbreviations ix

1 INTRODUCTION 1

1.1 Context . . . 1
1.2 Scope of Literature Review . . . 2
1.3 Data Gathering Costs . . . 2
1.4 Nonlinear Network Substitutions . . . 3
1.5 Linear Transform Substitutions . . . 3
1.6 Going Forward . . . 4

2 LITERATURE REVIEW 5

2.1 Introduction . . . 5
2.2 Architecture Scope . . . 6
2.3 Weight Importance . . . 7
2.3.1 Importance by Gradients . . . 8
2.3.2 Sparsity Inducing Penalties . . . 9
2.3.3 Latent Variable Methods . . . 10
2.3.4 Adding Noise and Variational Approximations . . . 11
2.3.5 Heuristics . . . 12
2.4 Quantisation . . . 14
2.4.1 Substituting Gradients for Quantisation . . . 14
2.4.2 Full Binary Networks . . . 16
2.4.3 Ternary Networks . . . 16
2.4.4 Fixed Point Precision . . . 16
2.4.5 Lossless Coding . . . 18
2.5 Network architecture . . . 18
2.5.1 Simplicity for Efficiency . . . 19
2.5.2 Grouping Convolutions . . . 20
2.5.3 Modulating Computation . . . 23
2.5.4 Low-Rank Approximations . . . 25
2.5.5 Efficient Linear Layers . . . 27
2.5.6 Distillation . . . 27
2.5.7 Architecture Search . . . 28
2.6 Information Theory . . . 29
2.7 Conclusion . . . 31

3 RESOURCE-EFFICIENT FEATURE GATHERING AT TEST TIME 35

3.1 Introduction 35
3.2 Methods 38
3.2.1 Noise Generating Processes 39
3.2.2 Related Work 42
3.2.3 Simultaneous Statistic Optimisation 43
3.3 Experiments 45
3.3.1 Digit Rotation Inference 45
3.3.2 Sensor Array Imputation 48
3.3.3 Learning Image Quantisation Matrices 51
3.4 Conclusion 58
4 EFFICIENT ARCHITECTURES 61
4.1 Introduction 61
4.1.1 Attribution 63
4.2 Related Work 63
4.3 Compression with Cheap Convolutions 64
4.3.1 Distillation 64
4.3.2 Cheap Convolutions 66
4.4 CIFAR Experiments 69
4.4.1 Network Descriptions 71
4.4.2 Analysis and Observations 73
4.4.3 Optimisation Dynamics 75
4.5 Scaling and Generalisation 77
4.5.1 ImageNet 77
4.5.2 Semantic Segmentation 80
4.6 Conclusion 81
5 COMPRESSED LINEAR TRANSFORMS 84
5.1 Introduction 84
5.1.1 Related Work 86
5.2 Methods 86
5.2.1 Methods To Compare 87
5.2.2 Separable Convolutions 88
5.2.3 Substitute Linear Transformations 88
5.2.4 Parameter Cost Grid Search 97
5.2.5 Training Compressed Linear Transforms 97
5.3 Experiments 99
5.3.1 Experiment Setup 100
5.3.2 Tensor Decomposition Settings 102
5.3.3 Parameter Use 102

5.3.4 Arithmetic Operations 107
5.3.5 Compression Ratio Scaled Weight Decay 112
5.3.6 Is Distillation Necessary? 113
5.4 Conclusion 115
6 CONCLUSION 117
6.1 Summary 117
6.2 Advances 119
6.3 Future Work 120
6.4 Final Comments 121
List of Figures 122
BIBLIOGRAPHY 130

ABBREVIATIONS

- **ARD** – Automatic Relevance Determination
- **AT** – Attention Transfer
- **CDF** – Cumulative Density Function
- **CIFAR** – Canadian Institute For Advanced Research dataset
- **CRS** – Compression Ratio Scaled
- **CUDA** – Compute Unified Device Architecture
- **cuDNN** – CUDA Deep Neural Network
- **DARTS** – Differentiable Architecture Search
- **DCT** – Discrete Cosine Transform
- **ELBO** – Evidence Lower Bound
- **FFT** – Fast Fourier Transform
- **FLOPs** – Floating Point Operations
- **GPU** – Graphics Processing Unit
- **HMC** – Hamiltonian Monte Carlo
- **IoU** – Intersection-over-Union
- **KD** – Knowledge Distillation
- **KL** – Kullback-Leibler
- **LSTM** – Long Short-Term Memory
- **MLE** – Maximum Likelihood Estimation
- **MLP** – Multi-Layer Perceptron
- **MNIST** – Modified National Institute of Standards and Technology database
- **mult-adds** – multiply-accumulate (operations)
- **NAS** – Neural Architecture Search

- **NLL** – Negative Log Likelihood
- **SGD** – Stochastic Gradient Descent
- **ST** – Straight-Through (estimator)
- **SVD** – Singular Value Decomposition
- **TRN** – Tensor Regression Networks
- **TT** – Tensor-Train
- **WRN** – Wide ResNet

1 INTRODUCTION

Machine learning consumes resources. Resources are either consumed in the gathering of data to supply a learning algorithm, or in the computations to process it. In this thesis we are concerned with how we can use fewer resources, either memory or the energy in computation, whilst still being able to use the flexible algorithms involved in modern machine learning.

1.1 CONTEXT

Historically, machine learning has not been focused on the efficiency of algorithms. Often, the hardware available at the time is pushed to achieve the best possible accuracy by using the maximum available resources (LeCun et al., 2015). In real applications, or on resource limited hardware, it is important to consider how our learning algorithms can operate using fewer resources. However, we must do this without losing the predictive performance that led us to use machine learning in the first place.

When applying a learning algorithm, the first major resource to be allocated is towards the gathering of data. The cost of gathering such data is a concern that should be included into the design of our learning algorithm.

For example, in the design of a sensor network, we must decide where to place sensors and how many sensors to place (Richman and Mannor, 2016). This is a decision that should depend on previously available data, and the model we build should be designed to incorporate the data gathered from new sensors. The data gathered once these new sensors are acquired and placed are referred to here as the "test time" scenario.

Alternatively, the learning algorithm could be designed to select features for predictions on the fly – while making a prediction. This presents a granular resource allocation problem, choosing where to allocate resources at each step. This may be relevant, for example, where features are the product of on-demand processing (Janisch et al., 2017; Contardo et al., 2016; Bojarski et al., 2016). In other words, when a feature is required, it will be computed. The problem is to design a predictive model that can operate with this constraint.

The model we produce should be flexible and scale to large datasets. This model could be composed using deep neural networks trained by stochastic gradient descent (SGD). In that case, the implementation of such a network should operate using a minimum of resources when deployed at test time on unseen data.

1.2 SCOPE OF LITERATURE REVIEW

In Chapter 2, we collate findings from the literature that report redundant computation in deep neural networks. Removing redundancies in deep neural network design therefore offers large efficiency benefits. For example, it has been demonstrated that only 10% of the connections in a network are required, and those that remain may be represented using only 5 bits each (Han et al., 2015).

We find deep neural networks are redundant in various different ways. As mentioned, neural networks are robust to removal of weights or units. The weights themselves may be stored at much lower precision. As network design has progressed, entire blocks that were previously thought to be vital, to a deep neural network, have been found to be unnecessary.

1.3 DATA GATHERING COSTS

This thesis is concerned with efficiency at test time, firstly the gathering of test time data and secondly in the design of the algorithm performing inference. Specifically, following contemporary trends in the field, we assume that the inference algorithm will be a deep neural network and focus on methods to reduce redundancy that are applicable over the widest possible range of such networks.

The first of our concerns, gathering test time data, is typically not considered when building a machine learning model. The model is trained on a training dataset and then applied at test time in some application. Resources spent obtaining test data are separate from those spent processing that data with the predictive algorithm.

In Chapter 3 we investigate how a model can be built that *does* take into account resources spent at test time. By explicitly modeling the changes we expect to see from using different test data, depending on the resources spent, we are able to build predictive models that perform better in that test setting and use fewer resources. We present methods to achieve this in flexible probabilistic models trained using stochastic gradient descent, on toy problems and real world data. Finally, by modeling quantisation noise we are able to automatically learn parameters used in JPEG compression (Wallace, 1991) that normally have to be tuned against human perception.

1.4 NONLINEAR NETWORK SUBSTITUTIONS

The second of our concerns involves the deep neural networks that are now common in conditional probabilistic models. To build more efficient predictive models, we must address the redundancy in such networks.

In order to incorporate an efficient deep neural network design in any predictive model, we would prefer that it operate in a familiar way. In other words, that it perform the same function and can be trained by the same algorithms we are already familiar with. To this end, we propose a way to modify an existing network to make it more efficient and, without tuning, provide a way to use existing training algorithms to produce a more efficient version of said network.

In Chapter 4 we present a protocol involving substitute convolutional blocks and distillation to produce efficient deep neural networks. The resulting efficient networks operate over a range of accuracy trade-offs against storage size or computational requirements. We find that the compression achieved is competitive with the state of the art and can generically be applied to any deep convolutional network. It is then demonstrated on small and large image classification problems, along with reducing the storage requirements of a semantic segmentation model to one quarter of the original, with little effect on the performance.

These results demonstrate that the constituent nonlinear transforms in deep neural networks are redundant, because they can be readily replaced without affecting performance. The parameters in these nonlinear transforms are stored in layers that implement *linear transforms* as a matrix multiplication between an input and a matrix of parameters. Other works, such as those discussed in Section 2.5.4, further demonstrate that *some* linear transforms in deep neural networks may be replaced by alternatives that consume fewer resources. Taken together, it is possible to ask, "can we replace *any* linear transform in contemporary deep neural networks with an efficient alternative?"

1.5 LINEAR TRANSFORM SUBSTITUTIONS

To provide a universal alternative to the linear transform used in deep learning, in Chapter 5 we compare methods that have been proposed in the literature to provide a resource constrained substitute linear transform. These substitute transforms are compared on known image classification benchmarks, being substituted into contemporary deep neural network architectures. This produces deep neural networks that can match the performance we expect from the best published results in the literature, while using considerably less resources.

1.6 GOING FORWARD

Machine learning can be thought of as providing predictions that may be of use to people. However, increasingly, the predictive task is subsumed under the task of *decision making*. Being able to account for a budget in a machine learning model can be useful in such settings, and the investigation in Chapter 3 could enable practitioners to reduce resource utilization.

In addition to this, this thesis provides methods to reduce redundancy in neural networks. Substitution of inefficient processing blocks in deep neural networks is sufficient to reduce resource requirements. In reducing these requirements, deep neural networks may be applied to resource limited or embedded devices. Operating on such devices may enable applications that are currently impossible, such as in remote sensing.

In this thesis we show that machine learning can be practiced while using an order of magnitude fewer resources, and with only small changes to the existing methods. We find that this is possible due to the well documented redundancy present in deep neural networks. Along with the progress towards cheaper computational resources, we expect that research into efficient machine learning will be transformational to any area using learning algorithms.

2 LITERATURE REVIEW

2.1 INTRODUCTION

Deep learning has developed empirically, with current best practices determined by experiment. AlexNet (Krizhevsky et al., 2012a), for example, was justified post-hoc in the choice of using convolutional layers, pooling and a specific structure, and has been very influential in future designs in deep learning. Many similar image classification problems have been approached using networks directly based on it. As new motifs win competitions, they are picked up by other researchers in the field; as an example, after ResNets (He et al., 2016a) were published, residual connections found their way into many future architectures (Zagoruyko and Komodakis, 2017; Zoph and Le, 2016; Liu et al., 2018).

In practice, on a new dataset, a machine learning practitioner can try to apply the architectures from the literature, but there is very little applicable theory to guide development. If the network does not converge on the dataset, it can be very difficult to diagnose what may be wrong: the dataset could be the issue, or the optimisation algorithm, or the architecture is badly suited to the problem. There is a clear lack of understanding in how these algorithms work (Goodfellow et al., 2016, p.416), and in this chapter we review results that make empirical measurement of that lack in understanding.

Other models used in machine learning are treated differently: Bayesian linear regression is tractable, exact and has a unique result given the data (Murphy, 2012, p.225). Random forests and other estimators are incorporated into fully automatic algorithms for many problems (Feurer et al., 2015). Successful applications of neural networks require an experienced practitioner with the time and resources to tune the algorithm.

More efficient algorithms for either learning or inference for Bayesian linear regression now depend on the optimisation of linear algebra calculations. In deep learning, the learning and inference algorithms are not explicitly derived and so cannot be abstracted away into solely linear algebra optimisations; despite the efficiency benefits any speed up of the linear algebra may bring to the fundamental computations.

The fundamental computations of deep learning may vary radically without compromising the learning algorithm. If we had a mature understanding of deep learn-

ing, it would not be possible to maintain performance by, for example, randomly setting half of all activations to zero (Srivastava et al., 2014). Similarly, there are many methods for sparsifying neural networks that are able to achieve equally good performance despite large algorithmic differences. These are compared in Section 2.3. Sparse neural networks suggest that the long optimisation routines of deep learning typically yield weights that may be inferred from a fraction of said weights (Denil et al., 2013).

Redundancy in the weights may suggest that the floating point precision used to store the weights is unnecessary. Many experiments demonstrate that networks using substantially fewer bits can operate, so common practice is wasteful. Reducing the precision by quantisation is discussed in Section 2.4.

How the network is structured, in number of layers and their composition, is treated carefully in practice. Researchers default to known "good" designs, but these rigid designs hide the many other possibilities. A common application of deep learning in research is also the one in which it has been profitable: image classification. In this domain the variety of architectures has become large as development is only constrained by experiment. In Section 2.5 we explore some of the variety within this domain.

The weights of a neural network must encode some information about the training set in order for that network to make useful predictions. Non-parametric methods, such as a Gaussian Process (Murphy, 2012, p.515), must process the entire training set to make a single prediction. Given that the size of many deep learning datasets exceed gigabytes, it may be surprising to note that the deep network learnt can only take up megabytes. Section 2.6 explores this issue, looking at the papers (MacKay, 1992; Hinton and van Camp, 1993; Shwartz-Ziv and Tishby, 2017) investigating the question of how this can be the case.

2.2 ARCHITECTURE SCOPE

In this review of the literature we focus on image classification architectures for two reasons: they are a popular application of deep learning (Goodfellow et al., 2016, p.326) and because they can be resource intensive (Real et al., 2017). In addition, the task has a clear end goal, and a common metric for grading success. Examples of accurate resource intensive architectures we are concerned with are: AlexNet (Krizhevsky et al., 2012a), ResNet (He et al., 2016a) and NASNet (Real et al., 2017). These are typically convolutional and can have hundreds of separate layers.

Most of the effort in efficient deep learning research has focused on these deep image classification architectures. For example, methods such as Deep Compres-

sion (Han et al., 2015) or specialised architectures such as MobileNet (Howard et al., 2017) are well known. Therefore, to simplify comparison and include most of the efficient deep learning research, we limit this review to deep image classification.

2.3 WEIGHT IMPORTANCE

Methods that reduce the redundancy of weights in neural networks by removing them either before, during or after training are reviewed in this section. Redundancy is intrinsic to deep learning. For example, it is common to train networks while removing units randomly. This is known as dropout and the results in a network whose predictions are robust to said removal (Hinton et al., 2012; Srivastava et al., 2014). While the test error rate reported in dropout experiments is typically with all units included, the train error is not, and the correspondence between the two demonstrates the redundancy of those units.

One improvement to randomly removing weights would be to evaluate the effects on the loss when a weight is removed. This would take an infeasible length of time to do exhaustively: evaluating the loss over a dataset for each weight would require as many passes on a validation set as weights in a network. Section 2.3.1 describes work on using backpropagated gradients to estimate the effect on the loss when removing weights. At any time, weights can be removed whose effect on the loss is minimal.

Removing weights at an arbitrary time may overlook the potential for *training a network to have few weights*. During network training an additional component to the loss can be added that encourages a network to be sparse. Published methods taking this approach (Hanson and Pratt, 1989; Han et al., 2015; Alvarez and Salzmann, 2016; Collins and Kohli, 2014) are described in Section 2.3.2. Adding terms to the loss can also be motivated probabilistically: the choice of prior can express whether the weight is likely to be near zero.

A more principled probabilistic approach is to consider the network through the lens of approximate inference: the network is then a latent variable model. The options explored so far for approximate inference are described in Section 2.3.3.

One type of approximate inference that has been successful is doubly stochastic variational inference. This approach, and the relevant papers (Kingma et al., 2015; Molchanov et al., 2017; Louizos et al., 2017; Federici et al., 2017; Gal et al., 2017) employing it, are described in Section 2.3.4. A factor in its success is that it can be implemented by the addition of noise during training, similar to common training routines using noise for regularisation.

Section 2.3.5 catalogues a collection of tricks that have been found to work in creating sparse neural networks. Typically, these arise from engineering insight when

interacting with the experimental side of deep learning, and do not fit into the categories investigated in other sections.

In this survey we focus on experiments involving image classification. Typically these experiments will focus on either the Imagenet dataset (Deng et al., 2009) or the CIFAR-10 dataset (Krizhevsky, 2009). Imagenet is a database of millions of images, typically processed at 256x256 pixels, while CIFAR-10 is a dataset of 60,000 much smaller images at 32x32 pixels. Experiments on ImageNet are much more time consuming than those on CIFAR-10 but are necessary to provide reliable results about whether a network can perform classification well at scale.

2.3.1 *Importance by Gradients*

Despite the universality of stochastic gradients for updating the weights in neural networks, it remains difficult to update the loss for the inclusion or exclusion of weights, thanks to the non-differentiability of the loss with respect to those variables. However, there are approximate gradient methods for optimizing the inclusion of weights.

Given a training or validation dataset, it is possible to remove any parameter from a network and observe the result on the loss function. It only requires running the network on every example of the dataset with that parameter removed. When we want to try all of the parameters, or any combination thereof, we require something less computationally intensive. LeCun et al. (1990) propose a Taylor approximation around the loss function, up to the second order derivatives:

$$\delta E = \sum_i g_i \delta u_i + \frac{1}{2} \sum_{i \neq j} h_{ij} \delta u_i \delta u_j + O(||\delta U||^3), \tag{2.1}$$

where E is the objective, $h_{ij} = \frac{\delta^2 E}{\delta u_i \delta u_j}$ are components of the Hessian, g_i are the gradients of the loss function with respect to U, the parameters, and δu_i is the proposed perturbation of the parameters.

Unfortunately, calculating the Hessian components, h_{ij}, is still expensive, so LeCun et al. (1990), and the following works discussed, employ approximations. The most common is a diagonal approximation: only evaluating h_{ii}, which can be calculated by backpropagating gradients from layer to layer with the same computational complexity as a first order backward pass (LeCun et al., 1990).

A recent application of this idea used a layer-wise application, pruning layers at separate points during training instead of trying to prune the entire network at the same time (Dong et al., 2017a). It compared well to contemporary results: maintaining

within 1% accuracy of the VGG network (Simonyan and Zisserman, 2014) on the ImageNet task with only 7.5% of the parameters remaining. However, the authors stress that the benefit of a layer-wise schedule to pruning helps to reduce the number of training iterations required *after pruning*, which can take as long as training the original model.

Computing the full Hessian is expensive, so other options have been explored to compute it approximately. Soon after LeCun et al. (1990) a method was developed to compute the full Hessian approximately (Hassibi and Stork, 1993). Since then, papers have focused on faster, less accurate approximations using only the gradients already computed in the backward pass (Theis et al., 2018; Molchanov et al., 2016).

Conceptual changes to the function of the algorithm have also been considered. It is possible to consider adding units while pruning, to actively modify a network, as was the approach of Guo et al. (2016). Or, units can be removed one at a time, while applying a renormalisation to the weights to maintain similar activations (Srinivas and Babu, 2015).

There is still mileage in Taylor expansion approximations for estimating changes to the loss function for each parameter, and using it to propose which parameters to remove: recently it has also been applied to network quantisation (Choi et al., 2016; Hou et al., 2016)(quantisation is discussed in more detail in Section 2.4). There is likely scope for more work in this direction, improving the approximation or using the information from it in a new way.

2.3.2 *Sparsity Inducing Penalties*

Making the training algorithm of deep learning any more computationally or intellectually intensive is often unsuccessful, so a popular method for sparsification is to incorporate a penalty term to the loss that will induce sparsity. The most well known of these methods is L1 regularisation, which adds the following term, $L1(U)$, to the loss (Hanson and Pratt, 1989):

$$L1(U) = \sum_i |u_i| \tag{2.2}$$

In practice this loss will gradually push parameters that are not used to zero, but weighting this penalty against the classification loss must be carefully tuned. Part of the training set is left out of training to use for tuning. This allows the designer to set the intended level of regularisation measuring loss on the validation set.

Alternatively, it is possible to use the Euclidean norm, or L2 norm, to create a penalty. These L2 losses are more commonly used for regularisation than L1 losses in neural network training. As the parameter gets close to zero the gradient of the L2 penalty also tends to zero, so these induce sparsity much more slowly, if at all. To enforce sparsity, one method could be to periodically set to zero any parameters that are below an arbitrary threshold. In Han et al. (2015) this was used to remove 92.5% of the parameters from a VGG-16 model, without losing any accuracy.

The sparsity produced by this type of regularisation not structured. Weight matrices contain arbitrary patterns of zeros and this sparsity is harder to exploit for efficiency than, for example, removing a row of a weight matrix. Alvarez and Salzmann (2016) propose a grouped penalty to prune entire filters from convolutional networks. They were able to reduce a network to 20% of its original size without affecting accuracy, observing 10-50% reduction in the time to process a minibatch.

Going a step further than L1 regularisation, Collins and Kohli (2014) consider engineering a penalty to mimic the L0 norm loss, which would involve penalising precisely the number of non-zero elements in a vector. This network used 14% of the weights of a pre-trained AlexNet, while the method of Han et al. (2015) was able to achieve 11%. Neither affected the accuracy of the network and the L1 regularisation of Han et al. (2015) is simple to implement compared to the details of the algorithm used by Collins and Kohli (2014).

Applying a regularising component to the loss function is a natural way to augment the training of neural networks but requires careful tuning. To find the correct settings for the hyperparameters weighting these regularisers, practitioners may use systematic grid searches, random search or black box optimisation. In either case, it is necessary to have a validation set and the ability to run a large number of experiments.

2.3.3 *Latent Variable Methods*

L2 or L1 regularisation can be derived as the consequence of Maximum Likelihood Estimation (MLE) in a linear model with a Gaussian or Laplacian prior on the weights. In contrast, we could make the assumption of a latent variable expressing the prior that most weights be zero. This is known as automatic relevance determination (ARD) (Neal, 1995; MacKay et al., 1994; Lawrence, 2001). It was developed in the context of learning the important units in a network (typically at the input). Neal (1995) pioneered this using Hamiltonian Monte Carlo (HMC) to perform inference.

Later work has made latent variable modeling in deep networks more tenable, but ARD has never seen large practical application. Recently, applying the most popu-

lar and scalable approximate inference methods for deep models, doubly stochastic variational inference, has been tested (Karaletsos and Rätsch, 2015). Unfortunately, the experiments do not focus on efficiency, focusing instead on the properties of the generative models being studied.

ARD places a specific form of sparsity inducing prior, focusing on removing irrelevant units. More general algorithms for inducing sparsity using doubly stochastic inference are the focus of the next section.

2.3.4 *Adding Noise and Variational Approximations*

Variational methods build an approximate posterior distribution to perform inference in a probabilistic model. This is made practical in neural networks by minimizing the Kullback-Leibler (KL) divergence between the approximation and the true posterior using the same stochastic gradient methods popular everywhere else in deep learning. To achieve this, a loss function called the evidence lower bound (ELBO) is derived. By sampling from the approximate posterior, an unbiased estimator of the gradient of the ELBO is produced, which minimizes the KL divergence between the true and approximate posterior (Kingma et al., 2015). In practice, this means adding normally distributed noise to units in the network, and backpropagating with respect to a loss that penalises the network to reduce that noise. The choice of prior becomes an important hyperparameter controlling whether the network will be sparse and other considerations.

Explicitly defining the prior over the weights in a neural network is the basis for Bayesian neural networks. The most popular direction in that research has become variational dropout (Kingma et al., 2015). With the addition of noise, and a penalty to reduce the noise added, the inference algorithm uses stochastic gradient descent (SGD) in the same way it is used in training a traditional neural network. Before this, it was necessary to sample the weights (Blundell et al., 2015), and other work applied similar noise but lacked the variational derivation (Nalisnick et al., 2015). Variational dropout initially did not produce sparse networks; it was intended only to be a cheap way to build a Bayesian neural network.

Molchanov et al. (2017) made a small change to the original variational dropout algorithm (Kingma et al., 2015) so the variance of the approximate posterior can tend to infinity. At training time, if the variance is high, that weight or unit is pruned. The experiments produced networks with greater sparsity than those of Han et al. (2015) without affecting performance. To deal with the same unstructured sparsity problem discussed in 2.3.2, a variant of this algorithm to induce structured sparsity was also developed (Neklyudov et al., 2017).

Unfortunately, both variational dropout algorithms (Molchanov et al., 2017; Kingma et al., 2015) use an improper prior, which makes Bayesian model comparison impossible. Using a proper prior produced an algorithm named Bayesian Compression (Louizos et al., 2017), which was used to prune nodes and apply a fixed point representation to the weights at the same time (Louizos et al., 2017; Federici et al., 2017). Unlike the fixed point representations in Section 2.4.4, the authors propose a variable number of bits depending on the weight tensor containing the weight. The results in this paper clearly demonstrate the power of variational methods. One result reported was 771 times smaller than the original network, which was a compression rate 20 times greater than that of Han et al. (2015).

As an aside, a more direct solution to the problem of learning a sparse structure would be to learn dropout probabilities for all units. Concrete dropout (Gal et al., 2017) provides a similar variational framework that would accommodate this scheme.

Variational algorithms for Bayesian networks have become the most popular method thanks to their relative similarity to training a network with dropout. While the results for compression have been impressive, the scope for inclusion in a larger Bayesian probabilistic model has been limited, despite this being one of the major attractions of bringing a neural network into the fold of Bayesian modeling. Fortunately, the efficiency improvements have been impressive, with the work of Louizos et al. (2017) achieving equal or better compression rates to other published methods, such as Deep Compression (Han et al., 2015) or Sparse Variational Dropout (Molchanov et al., 2017).

2.3.5 *Heuristics*

In deep learning a number of advances within the field are each motivated by a single engineering trick. These methods do not necessarily compose with other published work, or provide obvious directions for future research. However, they may be able to teach us something about how deep networks function, or point to the limitations of more popular methods.

Lebedev and Lempitsky (2015) show sparsification by fixing weights to zero at initialisation and the resulting networks performed as well as the full network. They go on to develop methods to choose fixed sparsity patterns. The fact that the networks learnt in this paper are comparable to those using more complicated pruning schemes could suggest that learning in deep networks is simply robust to sparse matrices, and not that pruning has to be done well. In addition, Crowley et al. (2018) demonstrates that sparse network structures trained from scratch perform better than pruned ar-

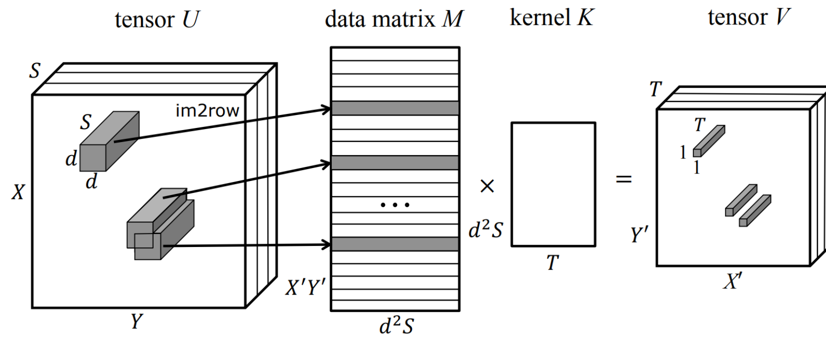


Figure 2.1: "Reduction of convolutional layer evaluation to matrix multiplication. Our idea is to leave only a subset of rows (defined by a perforation mask) in the data matrix M and to interpolate the missing output values."(Figurnov et al., 2016)

chitectures, lending weight to the argument that SGD is robust to sparsity in deep networks.

Ad hoc methods to prune whole filters from a convolutional network include pruning filters with small weights (Li et al., 2016), pruning according to energy usage (Yang et al., 2016) or tuning the sparsity regularisation, specifically a group Lasso (Scardapane et al., 2016), for a particular prescribed structure (Wen et al., 2016). All three of these methods get similar results despite the details of the implementations being different.

Another intuitive option that an engineer might consider would be to prune low-valued activations. Removing whole activations means no longer having to compute them, which speeds up inference (Albericio et al., 2016; Reagen et al., 2016). As with the other methods presented in this section, this can be achieved without affecting the accuracy of the resulting network.

Thinking carefully about the implementation of operations in a neural network can enable optimisations to be done. Figurnov et al. (2016) likely were inspired by considering that convolutions may be reduced to a matrix multiply between the data matrix of the input and the kernel function as illustrated in Figure 2.1. This led to perforated CNNs (Figurnov et al., 2016), which optimise sparsity in the input to each convolutional layer. Once the convolution is implemented as a matrix multiplication this becomes a mask on the input to that layer.

Heuristics may also be inspired by a change to the input distribution. Convolutional networks do not always have to process images. When processing data that is already sparse, such as 3D models or pen strokes, there are opportunities for sparse computation. Graham and van der Maaten (2017) re-engineer convolution to process these kinds of inputs efficiently, maintaining sparsity throughout the network. On the problems analysed, they achieve around a 50% speedup.

2.4 QUANTISATION

There are many units or weights in neural networks that can simply be removed (see Section 2.3), so it should perhaps not come as a surprise that it is unnecessary to know the weights to high precision. Many papers show that there are ways to reduce the precision of neural networks, despite the problems associated with dropping the assumption of continuous values for the stochastic gradient optimisation of the loss function. The degree of quantisation varies between papers, along with the adaptations to make optimisation converge, but the results are consistent. The exact values of weights in neural networks are not important, neither at training nor test.

The most extreme form of quantisation is to restrict the activations or weights to be only binary values. Stochastic optimisation is usually assumed to operate in a continuous space. In Section 2.4.1 we discuss how learning in deep networks can be made to work despite this assumption being broken.

Briefly, in Section 2.4.2 we mention work that has pushed networks to be binary in both activations and weights. These are rare and limitations are discussed.

Extra capacity can be added by allowing three values for each weight. These ternary networks are very similar to binary networks and are discussed in Section 2.4.3.

Increases in the possible values weights can take changes the problem significantly. The fixed point networks of Section 2.4.4 may have many more possible values, although still short of the massive space of floating point numbers. Their optimisation does not exhibit the same problems as those found in binary network optimisation. However, there are other problems, and the gains may be more limited.

Finally, a simple way to compress a network is to exploit some statistical pattern in the way it is stored using a generic algorithm. In the simplest case this could amount to lossless data compression (Welch, 1984) of the stored weight file to save space. Real applications of this technique are shown in Section 2.4.5.

2.4.1 *Substituting Gradients for Quantisation*

Considering only the value of the weights, the furthest we can push quantisation in a neural network is for the weight to only take two values. This is also a difficult optimisation problem, as the discretisation is non-differentiable, with no gradient on which to base the stochastic optimisation. Given no gradient, papers using binary weights have to resort to approximations. SGD only requires an estimate of the gradient as illustrated by the methods presented in this section.

If a binary weight is stored as a binary value, then it is extremely difficult to see how the updates could operate, with learning constrained to only update to one of

two values. To work around this problem the BinaryConnect algorithm (Courbariaux et al., 2015) proposed to store a continuous value with which to accumulate gradients. During training, this continuous value would be quantised to be either 1 or -1 on each pass.

The discretisation makes it impossible to evaluate the gradient with respect to the continuous weight. Without the true gradient, to update the weights we need something that provides an estimate of that gradient. BinaryConnect (Courbariaux et al., 2015) chose to use the straight-through (ST) estimator for this task.

In this case, the forward and backward passes are performed with a binary weight w_b sampled depending on the continuous stored w:

$$w_b = \begin{cases} +1 & \text{with probability } p = \sigma(w) \\ -1 & \text{with probability } 1 - p \end{cases} \tag{2.3}$$

However, we cannot evaluate the gradient with respect to the w weight variables. To work around this, the ST estimator uses the gradient with respect to w_b *in place* of the gradient of w (Bengio et al., 2013), yielding the following update rule:

$$w_t \leftarrow \text{clip}\left(w_{t-1} - \eta \frac{\delta E}{\delta w_b}\right), \tag{2.4}$$

E being the cost, or loss, function, η is a learning rate and clip is a threshold between -1 and 1. This update rule likely works because in expectation the discretised value is equal to the continuous. This same trick was applied by Courbariaux and Bengio (2016).

BinaryConnect (Courbariaux et al., 2015) can be applied at test time in a deterministic or stochastic form. In the stochastic many networks are sampled from the binary weights and the resulting ensemble used for prediction. Using the deterministic version XNOR-Net (Rastegari et al., 2016) demonstrated that this approach can work on ImageNet sized networks: AlexNet (Krizhevsky et al., 2012a), ResNet18 (He et al., 2016a) and a variant of GoogleNet (Szegedy et al., 2015).

To further speed up a network using this technique, particularly at training time, a quantisation trick can be used during backpropagation. Lin et al. (2015) propose quantizing the activations and error signal at each layer to a power of 2, so that the multiplication is just a bit shift. The speedup over a floating point multiplication is large, and the accuracy achieved on CIFAR-10 was not affected.

It is also possible to derive this type of algorithm from a variational Bayes perspective, as demonstrated by Soudry et al. (2014). The specifics are not covered in

this review, except to mention that the algorithm cannot match existing accuracy on convolutional networks.

2.4.2 *Full Binary Networks*

Soudry et al. (2014) propose a network which also quantizes the activations of the network, making the entire network a binary system. In this case, neural networks become boolean circuits, and learning them is an NP-complete problem (Pitt and Valiant, 1988; Kim and Smaragdis, 2016). Initial work on these could only learn extremely limited networks, such as those with only one hidden unit (Golea et al., 1993).

2.4.3 *Ternary Networks*

A natural extension to a binary network is to consider networks constrained to *three* values. In this setting the approximate gradient estimator of BinaryConnect (Courbariaux and Bengio, 2016) can be applied; and this has been demonstrated by Li and Liu (2016) and Ott et al. (2016); where it found a useful application in recurrent networks.

Ternary network performance can be advanced further by incorporating a trained scaling factor applied to the weights after quantisation (Zhu et al., 2016). These scaling factors can be grouped within weight tensors to further improve accuracy (Mellempudi et al., 2017), although this will come at a cost of more multiplications in the forward pass. In both cases training is by backpropagation, using estimators for the true gradient, which is not available.

After binary networks, ternary networks trade off a larger network size and more compute time for greater accuracy. To continue exploring this trade-off it is natural to consider networks using more than three values, but fewer than full floating point precision.

2.4.4 *Fixed Point Precision*

Fixed point precision refers to a custom data type using a signed integer with a fixed scaling factor, often used to represent numbers using fewer bits than 32 bit floating point. While this means fewer values can be represented, it reduces storage space and speeds up arithmetic. The research in this area is focused on the hardware

engineering challenges more than understanding deep learning, so this section does not describe any of these methods in detail.

The precision chosen could be a matter of hardware requirements, leading to methods that choose to constrain the weights to a fixed precision such as 8 bits (Ma et al., 2016; Gysel et al., 2016) or 3 bits (Venkatesh et al., 2016). In other cases, authors have derived the appropriate precision depending on the weight itself (Judd et al., 2016a; Moons et al., 2016; Louizos et al., 2017). It is also possible to combine variable bit length coding with alternative arithmetic schemes, such as bit-serial processing (Judd et al., 2016b; Moons and Verhelst, 2016). These have their own trade-offs, which are beyond the scope of this review.

Increasing the bit size further allows other training methods, including simply quantising the weights during (Lin and Talathi, 2016; Gupta et al., 2015; Hubara et al., 2018; Chai et al., 2017; Wu et al., 2015; Miyashita et al., 2016) or after training (Gong et al., 2014; Louizos et al., 2017; Zhou et al., 2017). DoReFa-Net (Zhou et al., 2016) adapts the straight-through estimator (Courbariaux et al., 2015) to operate with quantised gradients, for example at 6-bit fixed point precision, allowing training to be sped up along with inference.

While training networks with quantised weights is easier than binarised, there are some problems with stability if everything is simply moved to fixed precision. For example, gradient updates must be rounded to the precision of the weights. The resulting updates are a poor estimator for the true gradient updates. To deal with this, we can round up or down stochastically, with the mantissa as the Bernoulli probability. This technique is referred to as *stochastic rounding* and provides more reliable updates (Gupta et al., 2015).

Unfortunately, if rounding occurs on activations, even this can lead to large gradient mismatch problems as the error is backpropagated further through the network. One solution to this being to store full precision activations at training time (Lin and Talathi, 2016) as is typical in the binary networks of Section 2.4.1.

Fixed point precision incurs far less quantisation noise than the binarisation discussed in Section 2.4.1, so it should not be surprising that deep networks using these methods continue to function relatively normally. Deep networks trained with fixed point precision constraints possess similar efficiency benefits to other methods studied in this review. Networks may be stored using one tenth the number of bytes and run several times faster than similarly accurate networks.

2.4.5 *Lossless Coding*

An efficient way to store the weights is to take advantage of general methods for data compression. The simplest way to imagine this would be to apply Unix's "compress" utility to the stored weight file (Welch, 1984). However, unlike previous methods described for quantisation, this would not provide any benefit in computation time, or run time memory usage.

Lossless codes in general operate by providing a codebook matching codes to characters or patterns in a file. In the case of deep networks, we would be matching codes to repeating patterns in the weights stored. At inference time, the codebook matches codes back to the patterns, allowing the weights to be reproduced without error.

This is a good idea in applications. Deep Compression (Han et al., 2015) proposed a Huffman coding scheme to store weights in addition to pruning and quantisation. The combination was very effective, compressing a VGG-16 (Simonyan and Zisserman, 2014) network by 49x without affecting the accuracy it was able to achieve.

The numerical computation tools behind the rise of deep learning were developed for calculations in science and engineering that require more precision than neural networks apparently need in order to perform object recognition to high accuracy. Quantisation corrects this problem, showing that custom processors for deep learning have freedom to use inaccurate methods.

This section has shown that many of the architectures that were famous for winning classification competitions can be quantised (Welch, 1984; Han et al., 2015; Simonyan and Zisserman, 2014), and Section 2.3 showed weights could be removed. However, could this be a symptom of practitioners settling on redundant architectures? Can we design the architecture differently?

2.5 NETWORK ARCHITECTURE

Image classification is only one application of deep learning, but there is a surprising variety of network designs that can achieve good performance. We could imagine there is some space of possible networks that are able to perform the task, but the literature only explores a subset of it. In this section, we focus on networks designed with efficiency in mind. Often, these advances have been important and also found application elsewhere.

With experimentally verified designs researchers are pushed to try adding new ideas to networks. In Section 2.5.1 networks that try for *removal* of unnecessary components are discussed. For example, it has also been found that convolutional layers operating over all channels are often not necessary. Grouped convolutions decom-

pose these convolutions and have found wide application after being developed for efficiency, as described in Section 2.5.2. Convolutions are a key component of all deep learning work on images, so advances focused there can have wide consequences.

Neural networks are also commonly assumed to be a fixed feedforward system. Section 2.5.3 describes methods that break this assumption and are able to modulate computation depending on input or context. As an input propagates through either a modulated or fixed graph, the operations performed in it are typically defined in terms of matrices. Section 2.5.4 discusses efforts focused on decomposing the rank of these matrices for benefits throughout learning systems.

A low rank matrix explores only one way to speed up matrix operations. An alternative would be to fundamentally reparameterize the matrix operations. An area of research on *efficient linear layers* that do just this is discussed in Section 2.5.5.

All of these low level efforts, focusing on improving the elements that compose a deep network, are relatively independent to those looking at how to design the entire graph. Smaller networks use less resources than larger ones, but often there will be a cost in terms of performance. Distillation, covered in Section 2.5.6, provides a training regime for smaller networks without making that sacrifice.

Relying on special methods for training smaller networks may not be enough. What if we want a general way to find the smallest network good at a task? The topic of Section 2.5.7 is architecture search: the effort to automate the design of neural networks. These methods typically make explicit the implicit biases of neural network designers, showing what types of elements are expected in a network, while also showing unexpected ways to improve performance.

2.5.1 *Simplicity for Efficiency*

Fully connected layers were once popular in the final few layers of deep image classification networks, as illustrated in Figure 2.2a. One strategy to design an efficient network is to try and remove operations from an existing network, experimenting at each stage to check performance is not affected. Springenberg et al. (2014) provide an example of this: they remove the fully connected layers from a VGG-like (Simonyan and Zisserman, 2014) architecture and design a network to operate using only convolutions and pooling. The resulting network achieves performance matching more complicated designs, but it was not explicitly optimised for efficiency.

Design motifs focusing on convolution and pooling have become more widely adopted. ResNets, as illustrated in Figure 2.2b are a successful example of this. The recent rise of average pooling and pointwise convolutions in place of a fully connected layer massively reduces operations and parameters when training a simple convolu-

tional network, without affecting accuracy (Szegedy et al., 2015; He et al., 2016a; Lin et al., 2013). In fact, recent work has demonstrated no benefit to having parameters in the final layer; it can be replaced with a Hadamard matrix (Hoffer et al., 2018).

SqueezeNets (Iandola et al., 2016) are one of the most successful applications of a simple strategy. The network architecture used in their paper follows a sequence of rules intended to reduce the number of parameters used while maintaining representational capacity. These rules are to use 1x1 filters instead of 3x3 as much as possible, reduce the number of channels in hidden layers and keep activation maps large until close to the classification layers. Training this network architecture and then applying quantisation and sparsification led to the best performance on network compression reported at the time: SqueezeNet could be stored in 0.5MB and classified ImageNet with a top-1 error of 42.5%. Despite this small storage size, the computation required to make predictions with this network is relatively high, illustrating one of the key design considerations when approaching efficient deep neural networks. The number of operations used by this architecture is higher than larger models: MobileNet-1.0 (Howard et al., 2017) has approximately 4 times as many parameters but uses 3 times fewer mult-adds.

2.5.2 *Grouping Convolutions*

Running independent convolutions on tensors split evenly along the channel dimension is a method to improve efficiency going back to the original AlexNet paper (Krizhevsky et al., 2012a). Although, in that case it was for efficiency in training. Grouping by channel has now become so common it is a part of cuDNN (Chetlur et al., 2014) (CUDA deep neural network). Which, along with CUDA (compute unified device architecture), forms the most common backend for deep learning frameworks.

In Figure 2.3 the channel-wise split used in a grouped convolution is illustrated. The input tensor is split into groups of channels, independent filters are passed over each of these groups, and the groups are then concatenated together again. As the number of groups increases less parameters are used. Convolution uses parameters quadratically with the size of the channels. Performing independent convolutions over channel groups therefore uses fewer parameters. In the extreme, when there are as many channels as groups, the grouped convolution only uses parameters linear in the number of channels.

Having the same number of groups as channels is the most limiting form of grouped convolution; information in any channel cannot influence another. To deal

(a) The AlexNet architecture (Krizhevsky et al., 2012a) (cropping is true to the original paper).

(b) ResNet architectures (He et al., 2016a) compared against VGG-19(Simonyan and Zisserman, 2014).

Figure 2.2: More recent architectures are deeper and have skip connections, as in b, but they also rely on repeated 3x3 and pooling, in the style of Springenberg et al. (2014). Further, fully connected final layers like those in a are now replaced by average pooling as the final operation. This works thanks to strided convolutions reducing the size of the activation map at stages throughout the network.

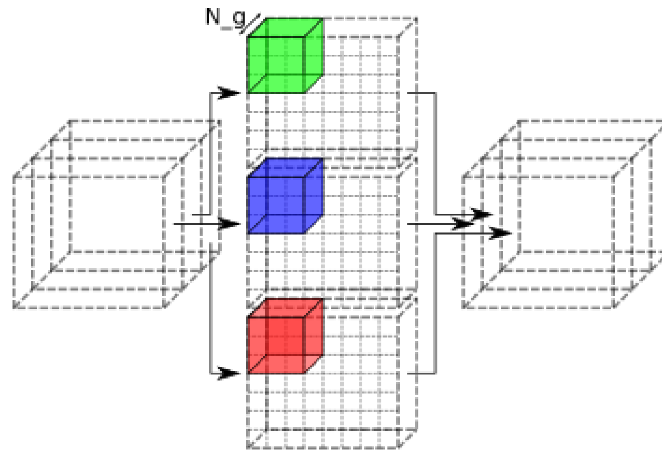


Figure 2.3: Grouped convolutions apply independent filters over an input grouped by channel.

with this, a common solution is to use a pointwise convolution after the grouped. The combination of the two is commonly called a separable convolution.

Laurent Sifre developed separable convolutions in their current form (Sifre, 2014). After this, they were used in the Xception architecture (Chollet, 2016) for improved results in classification – at the same time speeding up inference at test time. The most significant application for efficiency has been their application in the MobileNet architecture (Howard et al., 2017) and in the Inception block (Ioffe and Szegedy, 2015). MobileNet demonstrated separable convolutions for efficiency, achieving greater accuracy than SqueezeNet (Iandola et al., 2016) while using 22 times fewer multiply-add operations. Using a number of groups not equal to the number of channels is less popular, but is explored in some papers, such as Ioannou et al. (2016).

Separable convolutions are similar to intra-channel convolution (Wang et al., 2016). However, in this case the *same filter* is applied to channels independently, and then channels are linearly combined, repeating the process for as many output channels are required. Despite making a parallel operation sequential, the technique has been demonstrated to make ResNets (He et al., 2016a) use more than 4 times fewer FLOPs.

Another step to reduce parameter usage would be to group the pointwise convolution, but doing this would make channel groups disconnected throughout the network. To work around this problem, ShuffleNet (Zhang et al., 2017a) proposed a riffle shuffle of the channels in between alternating grouped pointwise convolutions. On mobile devices this network was more than twice as fast as MobileNet (Howard et al., 2017) within 3% error.

Deconstructing convolutions further for efficiency has also been considered in the literature. 3D convolutions along the channels and spatial axes of the input tensor can be as effective as a traditional convolutional architecture (Jin et al., 2015). Unlike most ways to modify the elements of neural networks, this approach shows a network

without the spatial kernel receptive field we have come to expect of deep learning, and shows that it can still classify natural images.

2.5.3 *Modulating Computation*

Typically, neural networks are feedforward structures, a fact reflected in the symbolic computation graphs built by many frameworks (Abadi et al., 2015; Al-Rfou et al., 2016). Fixed computation graphs allow for an input to pass through a series of immutable operations and for those operations to be optimised for efficiency. Breaking this assumption can help us discover significant efficiency gains. Computation can be modulated through dependence on the input or by scenario. This includes networks that are designed for anytime inference, improving predictions as more time is allocated at test time (Huang et al., 2017a).

Early work on deep learning found this idea attractive, as computing resources at the time were limited compared to today. LeCun et al. (1998) developed a three-stage boosting algorithm for classification; combining three LeNet-4 networks and outperforming a single LeNet-4, along with a support vector machine and tangent distance classifier; though, they did not analyse the energy efficiency.

Networks incorporating this design strategy have recently been used to push the ImageNet benchmark further. Squeeze-and-Excitation (Hu et al., 2017) use the globally pooled spatial statistics to modulate the convolutions performed in a ResNet (He et al., 2016a). If viewed as parameter sharing between different possible convolution operations, this is similar to the double recursive convolutions of Zhai et al. (2016). While neither paper was aimed at efficiency, performance was significantly improved at a small overhead from the original network. The computations were modulated depending on the input but both are still feedforward networks.

An early paper aimed directly at conditional computation for efficiency was Almahairi et al. (2015). The authors design a system using two types of networks, either low- or high-capacity to use depending on the results of a custom attention mechanism. On small image datasets they were able to show it worked, but it was not applied to larger benchmarks, such as ImageNet. This network is not strictly feedforward. Inputs are routed to different subnetworks, which are feedforward, depending on the output of the attention network.

Breaking the fixed feedforward assumption on ImageNet was known to be possible from the results of stochastic depth (Huang et al., 2016b), in which a ResNet was trained with a random variable number of blocks between input and output. This was later used for pyramidal residual networks (Yamada et al., 2016). Thanks to the shortcut connections used in ResNets, the output could always remain connected and

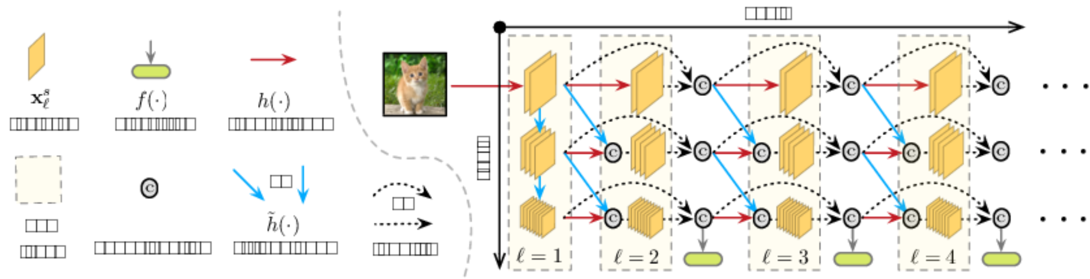


Figure 2.4: “Illustration of the first four layers of an MSDNet with three scales. The horizontal direction corresponds to the layer direction (depth) of the network. The vertical direction corresponds to the scale of the feature maps. Horizontal arrows indicate a regular convolution operation, whereas diagonal and vertical arrows indicate a strided convolution operation.” (Huang et al., 2017a)

the training strategy only yielded more regularisation. However, the authors did not investigate efficiency benefits at test time.

Those working on stochastic depth (Huang et al., 2016b) went on to develop multi-scale dense networks (Huang et al., 2017a), based on earlier work with the DenseNet architecture (Huang et al., 2016a). By having multiple output locations, the network can classify after variable processing times. The authors demonstrate how this can be trained efficiently by incorporating previous computation at every stage, as illustrated in Figure 2.4. The major competitor for anytime prediction was FractalNet (Larsson et al., 2016), which demonstrated a recursive blockwise structure in which an input could take an exponentially large number of different paths. At low mult-add budgets, multi-scale dense networks are around 1% less accurate than FractalNets, but as budget increases they are able to gain 10% in accuracy over all other competing methods. This could be thanks to DenseNets (Huang et al., 2016a) already being a competition winning architecture.

Dong et al. (2017b) present low cost collaborative layers that could provide a more generic structural modification. The idea is to use a simpler layer to predict where the activation map of the main convolutional layer will be non-zero, and then only perform calculations there. Acceleration in inference of around 32% is achieved, which is relatively small compared to most methods reviewed here. For example, multi-scale dense networks achieve greater accuracy, and in the range of a 32% speed up the accuracy does not change (Huang et al., 2017a).

Another way to consider the problem of conditional computation could focus on maintaining a state of the activations in a network and performing consistent operations to update that state towards a more accurate prediction. O’Connor and Welling (2016) develop an approach to this based on propagating quantised differences through a network. On video problems, the state can be propagated between frames, allowing this type of approach to show substantial efficiency benefits, but

the price of discretisation is high. Using a VGG-19 (Simonyan and Zisserman, 2014) architecture was reported to be difficult.

Conditional computation can be taken to an extreme by considering a network that uses different weights for every incident example. Two papers report success in this area: Hypernetworks (Ha et al., 2016) use a separate network to generate the weights for the network doing the task and Fast Weights (Ba et al., 2016) focus on modulating long short-term memory (LSTM) weights during propagation, based on the input. Hypernetworks are a more general abstraction, being applied to problems beyond sequence modelling, but the added abstraction and instability appears to have been a barrier to widespread adoption. However, they have been used for architecture search, allowing rapid generation of the weights to test an architecture (Brock et al., 2017).

2.5.4 *Low-Rank Approximations*

Using matrix decompositions, such as the Singular Value Decomposition (SVD), it is possible to make the matrix multiply efficient, if the weight matrix is low rank. For this reason, many papers have focused on methods to learn networks with low rank weight matrices. An early paper taking this approach, along with other approximations, was able to speed up convolutional layers by 1.6x (Denton et al., 2014) while reducing each SVD decomposed layer to be several times smaller than the original.

Convolutional layers in networks are banks of convolutional filters, and as such it is possible to compose these filters from rank-1 matrices. Rigamonti et al. (2013) describe a procedure for arbitrary filter banks, while Mamalet and Garcia (2012) develop a similar method specifically for deep learning.

Jaderberg et al. (2014) also consider rank-1 decompositions of the convolutional filters. By incorporating a loss based on the original networks activations, in the style of Section 2.5.6, it was possible to maintain performance within 1% in accuracy while achieving a 4.5x speedup. This method was extended in Tai et al. (2015) to improve the calculation of the tensor decomposition, training the network and evaluating on the ILSVRC12 (ImageNet) dataset (Deng et al., 2009). A similar rank-1 decomposition is used by Lebedev et al. (2014) but the authors do not use the original network for the retraining, using only the original training data. Nevertheless, the speedup of around 4x was still observed. A similar reduction in the number of FLOPs, by 4.93x on VGG-16, could be achieved by Tucker decomposition but only around a 2x speedup came with this (Kim et al., 2015).

AlexNet (Krizhevsky et al., 2012a) and VGG-like (Simonyan and Zisserman, 2014) architectures included most of their parameters in the fully connected layers, leading

to some papers focusing on only low-rank approximations to those. The Tensor-Train (TT) decomposition (Novikov et al., 2015) was able to achieve a 200,000 times smaller weight matrix in the final fully connected layers. This should not necessarily be surprising, as later network designs were able to remove the fully connected layers altogether (Springenberg et al., 2014). Although the technique of Novikov et al. (2015) is still powerful, a later work extended it to larger tensors used in convolutions and was able to reduce the size of a model on CIFAR-10 (Krizhevsky, 2009) by 80x while losing 1% accuracy (Garipov et al., 2016).

Unfortunately, implementing matrix-vector multiplication in an efficient way using TT-decompositions has not been demonstrated in a deep neural network. To work around this problem, one paper suggests using an alternative to a linear transform that is more appropriate when using tensors. Kossaifi et al. (2017) introduce Tensor Regression and Contraction Layers, defined to manipulate Tucker decompositions of high rank tensors. Starting with pre-trained ResNet (He et al., 2016a) and VGG (Simonyan and Zisserman, 2014) networks, the authors substitute such layers in place of the final pooling and linear layers for their experiments. They demonstrate 90% reduction in parameters used while maintaining accuracy within 3.5%.

All of these low-rank approximations consider maintaining the linear transformation while reducing the rank of the matrix. In a deep network there are typically nonlinearities between each linear transformation. Zhang et al. (2015) observe that the transform including the nonlinearity may be easier to approximate, for example if a nonlinearity sets an output to zero. They then develop an algorithm to produce a low-rank nonlinear transform close to the original.

Low-rank compression strategies typically apply to networks that have already been trained conventionally or are a hard constraint on the parameterisation of weights. Constraints can easily upset the delicate balance of stochastic optimisation in deep learning. A regulariser is often a safer choice, and the regulariser of Alvarez and Salzmann (2017) encourages weights to be easily decomposed to a lower rank matrix. This allows the authors to achieve approximately 1% greater accuracy than Denton et al. (2014) while using one tenth of the parameters.

After learning a low-rank approximation, the network can be stored more efficiently and there will be some computation speedup. Rakhuba and Oseledets (2014) move both the input and low-rank approximation into the frequency domain to make low-rank approximations even faster. However, they do not present experimental results. FFT-based convolution promised faster computation time, thanks to the convolution becoming an elementwise product (Mathieu et al., 2013) in the frequency domain, but numerical issues advised against its inclusion in the popular cuDNN library (Chetlur et al., 2014).

2.5.5 *Efficient Linear Layers*

Efficient linear layers reduce the problem of efficient architecture design to that of rethinking the matrix multiply involved in a linear transformation. Convolution can also be implemented efficiently as a matrix multiplication (Lavin and Gray, 2016), making these approaches general enough to consider in designing an efficient network. These are similar to the methods described in the previous Section, but we make the distinction that these layers substitute existing layers in deep neural networks without modifying the optimisation algorithm.

Deep Fried Convnets (Yang et al., 2015) approximate a matrix multiplication using a Fastfood transform. The Fastfood transform is composed of permutations, Hadamard transforms and diagonal random matrices, which are the trainable parameters. The advantage of this sequence of transformations is that the number of operations scales loglinearly instead of quadratically. Later, the same group developed a similar method using the discrete cosine transform (DCT) named ACDC (Moczulski et al., 2015), which simply applies two diagonal matrices of parameters between a forward and reverse DCT. ACDC could operate twice as fast as a Deep Fried Convnet (Yang et al., 2015), speeding up an AlexNet (Krizhevsky et al., 2012a) six times with only a 0.6% drop in accuracy.

"Structured Spinners" (Bojarski et al., 2016) have been proposed as a more general method based on a sequence of Hadamard and random diagonal matrices. This paper did not present results on large-scale image classification problems. As in all of these methods the particular parameterisation affects the convergence of SGD, which can be a barrier to adoption.

Despite the implementation of convolution in most frameworks as a matrix multiply, none of these papers attempt to reduce the number of convolutional parameters. Even at the time of their writing, architectures with mostly convolutional structures comprise all of the winning entries in major competitions (He et al., 2016a). It is likely that this change would affect the convergence of SGD even more, which could be the major reason keeping the unstructured matrix on top in deep learning.

2.5.6 *Distillation*

As the relationship to the number of parameters in a network and the learning capacity is not well understood we could choose a much smaller convolutional network architecture to save computation and storage. Unfortunately, we would typically find that this model does not perform as well as the larger model. However, if the smaller model is trained on the output of the larger model, it can perform just as well on test

data (Ba and Caruana, 2014). This has been called a teacher and student network or model distillation (Hinton et al., 2016). Bayesian scenarios (Korattikara Balan et al., 2015) and reinforcement learning (Parisotto et al., 2015) have also applied the same technique.

Following the discovery of limitations in the original model distillation method (Romero et al., 2014; Urban et al., 2017), interest in this method waned. Modern deep convolutional architectures could not be compressed into a less deep student architecture (Romero et al., 2014). However, a method called *attention transfer* using supervision at intermediate layers has demonstrated it is still possible to learn simpler ResNets that do the job of larger ones (Zagoruyko and Komodakis, 2017). Using this same loss, a way to distil a large network while allocating minimal engineering time is to map the larger network to a smaller by a reducing the size of repeated motifs, such as reducing the channels in convolutions (Crowley et al., 2017)(and this will be described in full in Chapter 4).

Some theory has been developed to explain how this is possible, and is explored by Lopez-Paz et al. (2015) and Vapnik and Izmailov (2015). The simplest explanation is that the information content of the logits of a trained network is much higher than the information content of a one-hot categorical vector, so it provides better supervision from which to learn. However, this is likely insufficient, because it is possible to train a group of student networks jointly, without the need for a teacher and obtain better results than if any of the networks had been trained individually (Zhang et al., 2017b). The network morphisms of Chen et al. (2015) (also applied by Elsken et al. (2017)) can also be viewed as a kind of model distillation in which the added capacity is trained using the predictions of the smaller model, turning the distillation idea upside down.

2.5.7 *Architecture Search*

Designing networks is typically performed by hand by the researcher, building on some known "good" architecture. Instead, some researchers have attempted to design algorithms that automatically design a network for a given application. Usually, this means explicitly encoding researcher's intuitions about what makes a network "good".

Neural architecture search (NAS) provided the first application of architecture search to large-scale image recognition (Zoph and Le, 2016). The method uses an LSTM to design network modules sequentially which then compose networks. After training the networks, the loss achieved on a validation set is used as a reinforcement learning signal to update the LSTM. The resulting NASNet had an efficient variant

intended for mobile networks and achieved state of the art accuracy at time of release. However, the paper received criticism for the extreme cost, reportedly using 450 GPUs for 3-4 days. The authors later released a paper using shared parameters between proposed designs that used 1/1000 of these resources, achieved by sharing parameters between candidate networks (Pham et al., 2018).

Training a network to design another network with a reinforcement learning loss is a relatively complicated task and, given the difficulty in making deep reinforcement learning algorithms converge, may be impractical. It would be more attractive to have a fixed algorithm that does not require tuning. A hill climbing algorithm would involve training the network, adding capacity somewhere, then training again and taking the step to that configuration if performance had improved. Using a cosine annealed learning rate and restarting the learning rate between changes to the architecture can make this program very fast and produce architectures that match those of NAS in only a day's processing on one GPU (Elsken et al., 2017).

Building a network with a hill climbing algorithm and increasing capacity throughout at random can produce a variety of network designs. NASNet, in contrast, performs a search over possible block structures then repeats this block identically throughout the network (Zoph and Le, 2016; Real et al., 2017).

While these methods seem effective in designing good image classifiers, this may only be because the search space defined by the algorithm is heavily biased to producing *known* good designs. The algorithms are always defined to use 3x3 and pointwise convolutions in places where researchers would expect them. In NAS, for example, the first step is always a downsampling convolution, which is typical of networks since AlexNet (Krizhevsky et al., 2012a)(although, one algorithm with more freedom was able to rediscover convolution, to some extent (Fernando et al., 2016)). Therefore, it is unlikely that these algorithms give much insight into the correct way to design a network architecture or to strip unnecessary processing in order to run an efficient deep learning algorithm. Some evidence that the problem of learning a network is constrained enough to make it easy to explore is given by the similar performance of random search in this space (Li and Talwalkar, 2019).

2.6 INFORMATION THEORY

This review has so far presented the many possible designs of deep networks, showing redundancy in common architectures. There are papers studying the redundancy in neural networks directly. For example, analysing how much a network may learn from the massive datasets to which they are exposed. From this work, it could then

be possible to say how much compression we ought to expect to be able to achieve, or how this depends on dataset size.

Neural networks could be interpreted as a parametric conditional probabilistic model, but the assumptions made in their design are limiting. Specifically, once learnt, the parameters used in applications are a point estimate, which does not permit uncertainty over logit distributions or Bayesian model comparison (MacKay, 1992). However, placing a proper prior on the weights would expose us to the problem of inferring the posterior from data, which has typically been more difficult than optimisation of the point estimate.

A posterior distribution over the weights of the network would also allow us to calculate the information carried in the model. This lead early papers to focus on ways to approximate the posterior distribution over the weights (Hinton and van Camp, 1993). However, to update the posterior, both the mean and variance of the normal must be kept track of and at the time this was not practical. Later work showed that small networks could be learnt by maintaining forward and backward estimates of probabilities in a similar way (Hernández-Lobato and Adams, 2015).

A variational approximation of the posterior was the focus of Bayesian neural networks discussed in Section 2.3.4, which is another way to express bits-back coding (Hinton and van Camp, 1993). Both are optimal when the variational distribution is equal to the true posterior (Honkela and Valpola, 2004). Bits-back coding was initially attempted with a mixture of Gaussian prior, but the method for approximate inference involved explicitly tracking mean and variance through the network (Hinton and van Camp, 1993). Later work adapted the variational dropout algorithm (Kingma et al., 2015) to develop an effective compression scheme using a mixture of Gaussian prior (Ullrich et al., 2017). It was able to outperform Han et al. (2015) on a toy digit classification task but could not be scaled to larger problems.

Looking at the weights learned in convolutional filters, many papers have commented on their resemblance to Gabor filters (Yosinski et al., 2014). Motivated by the spatial smoothness of these filters, Denil et al. (2013) developed a method to predict the filters in a network by training a kernel ridge regression on a smaller weight dictionary. It was possible, using the networks at the time, to predict all the remaining parameters in the network from just 5%. This matches the observations of papers in Section 2.3 as many report being able to remove at least as many parameters without a drop in accuracy.

From the binary quantisation schemes described in Section 2.4, we could conclude that we only require one bit per weight in the design of neural networks. However, it turns out it is possible to take this further, using an adaptation to the BinaryConnect (Courbariaux et al., 2015) involving non-linear distortions. Merolla et al. (2016)

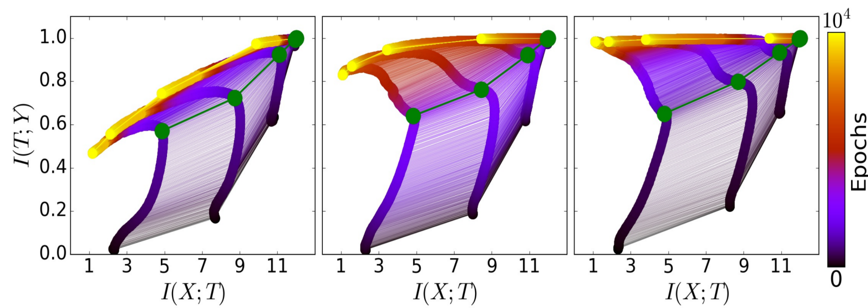


Figure 2.5: "The evolution of the layers with the training epochs in the information plane, for different training samples. On the left - 5% of the data, middle - 45% of the data, and right - 85% of the data. The colors indicate the number of training epochs with Stochastic Gradient Descent from 0 to 10000. The network architecture was fully connected layers, with widths: input=12-10-8-6-4-2-1=output." (Shwartz-Ziv and Tishby, 2017)

found only 0.68 bits per weight were required to train a state of the art network on CIFAR-10.

Investigating the mutual information between data and activations has gained popularity through the analysis of Shwartz-Ziv and Tishby (2017). Shwartz-Ziv and Tishby (2017) show dynamics in the mutual information between activations at different layers during the learning process of a deep network as shown in Figure 2.5. These figures plot the time series of mutual information between a layer's activations T, input X and Y in a typical classification problem using a neural network of 11 layers. The effects of overfitting, for example, can be seen in the leftmost plot: as training progresses the mutual information between all activations and the target Y drops.

They show that, even in early layers of the network, the training process proceeds towards throwing away the information about the input except that which is useful for the targets. While this does not provide a full explanation of deep learning, the authors suggest that deep networks benefit from the extra layers when training the network. As the number of layers increases training becomes faster.

Unfortunately, the scale of the networks in the work of Shwartz-Ziv and Tishby (2017) are orders of magnitude smaller than those in real applications. Therefore, the insight we can gain from this analysis is difficult to apply when inventing design strategies.

2.7 CONCLUSION

Deep convolutional networks are robust to many disturbances. The literature focusing on efficiency has been reviewed in this chapter. Despite growth in this field of research, we cannot explain precisely why many of the differences cause a network

to perform better or worse. Research on efficiency allows us to question which elements of a deep neural network are necessary and may help us to improve them.

Efficient machine learning is on the rise. Most of the papers in this review have been published since 2014. This is corroborated by previous review papers focused on the same topic (Sze et al., 2017).

The work in this growing field has been separated into work that focuses on sparsification, quantisation and architecture design. In many of the cases reviewed, the authors reported a 10 times compression factor, be it a compression ratio or test time speedup. As each method was able to achieve this, and each achieves this using a different approach, deep neural networks are certainly a redundant algorithm.

Direct comparison of the methods extant in efficient machine learning is difficult. The aims of works often differ; for example, the focus may be on storage efficiency, run time or number of mult-add operations. Also, methods may focus on different application areas. To give a summary of the state of the art, we summarise the achievements of seminal works in each section of this review: In the following top-1 error is reported on ImageNet (Deng et al., 2009), unless otherwise noted:

- Section 2.3.1: Fisher pruning reduces the FLOPs required to perform a gaze prediction task by 96.5% (Theis et al., 2018). It was also shown the same method would reduce the number of parameters of a ResNet-34 by 50%, while decreasing the accuracy 10% (Crowley et al., 2018).
- Section 2.3.2: A Group Sparsity (GS) regulariser allows a network achieving 31.9% top-1 error to remove 48.28% of the original 10.2M parameters (Alvarez and Salzmann, 2017; Alvarez and Salzmann, 2016).
- Section 2.3.3: Automatic Relevance Determination may be used to reduce the number of latent variables in a generative model of Frey Faces to 9, down from 30, while improving the marginal likelihood (Karaletsos and Rätsch, 2015).
- Section 2.3.4: Bayesian Compression (Louizos et al., 2017) can reduce the number of parameters in a VGG network by 94.4%, while increasing the error on a CIFAR-10 inference problem by 0.6%.
- Section 2.3.5: Perforated CNNs (Figurnov et al., 2016) demonstrate a VGG-16 network may perform inference twice as fast, while increasing error by 2.5%.
- Section 2.4.1: Using the straight-through estimator, it is possible to train a version of GoogleNet (Szegedy et al., 2015) with binary weights to within 5.8% of the accuracy achieved by the full precision network (Rastegari et al., 2016).

- Section 2.4.2: Fully binary multilayer neural networks are equivalent to discrete belief networks, and Expectation Backpropagation (Soudry et al., 2014) (EBP) has been shown to work well on small classification problems, but cannot be compared to the large image problems in other Sections of this review.
- Section 2.4.3: Ternary networks using Fine-Grained Quantization (Mellempudi et al., 2017) (FGQ) have been demonstrated to reduce the inference time of a ResNet-50 by 9x, while increasing the top-1 error from 24.95% to 29.24%.
- Section 2.4.4: Deep Convolutional networks are robust to fixed point weight precision; it has been demonstrated that AlexNet (Krizhevsky et al., 2012a), VGG networks (Simonyan and Zisserman, 2014), GoogleNet (Szegedy et al., 2015) and ResNets (He et al., 2016a) may be trained to within 1% the full precision top-1 error with weights stored using 8 bits (Zhou et al., 2017).
- Section 2.4.5: Deep Compression (Han et al., 2015) demonstrated a network with the accuracy of AlexNet Krizhevsky et al. (2012a) which could be stored in 6.9MB, 35x smaller than the original 240MB.
- Section 2.5.1: MobileNet Howard et al. (2017) denotes a set of networks achieving a loglinear range between 50% top-1 error/30M mult-adds/0.5M parameters and 29.4% top-1 error/569M mult-adds/4.2M parameters.
- Section 2.5.2: ShuffleNet Zhang et al. (2017a) denotes a set of networks achieving a loglinear range between 43.2% top-1 error/38M mult-adds/0.99M parameters and 24.7% top-1 accuracy/527M mult-adds/7.5M parameters.
- Section 2.5.3: Multi-Scale DenseNets (Huang et al., 2017a) can budget computation in a loglinear range of 500M to 3G mult-adds, achieving between 36.5% and 24.1% top-1 error.
- Section 2.5.5: ACDC (Moczulski et al., 2015) allows the final layers of a CaffeNet architecture to be compressed, removing 83.3% of the 58.7M parameters and increasing the top-1 error from 42.59% to 43.26%.
- Section 2.5.6: Moonshine (Crowley et al., 2017) compress a ResNet34 (He et al., 2016a), increasing the top-1 error from 26.73% to 30.16% while decreasing the parameters used from 21.8M to 3.1M and the mult-adds from 3.669G to 559M.
- Section 2.5.7: PNASNet (Liu et al., 2017) is an efficient architecture that achieves a top-1 error of 25.8% using 5.1M parameters and 588M mult-adds.

While we see that orders of magnitude improvements against chosen baseline architectures are often possible, many of the reported algorithms are still outperformed by

simple efficient architectures. For example, ACDC (Moczulski et al., 2015) allows for massive compression of an AlexNet (Krizhevsky et al., 2012a), but the performance achieved is still outperformed by MobileNet (Howard et al., 2017) using less than a fifth of the parameters. This leaves open questions about the applicability of existing methods to improve efficiency to newer efficient architectures.

This review therefore informs the investigation of Chapters 4 and 5. In both of these, we aim to *keep the training routine the same* while providing a more efficient architecture. In order that it be applicable to any new network structure proposed, it provides a way to modify any existing network. We show by experiment, in both chapters, the efficiency benefits achieved in this way, by substitution into competitive networks.

3 RESOURCE-EFFICIENT FEATURE GATHERING AT TEST TIME

3.1 INTRODUCTION

Machine learning systems consume resources in order to produce predictions. These resources may be the energy required to process data, train a learning algorithm, compute a prediction, or they may be the resources required to gather the data to supply such an algorithm. In this chapter we consider how to include these considerations in a probabilistic model in order to make the most accurate predictions we can, given a budget.

When we allocate resources from a budget, we are going to affect the quality and quantity of the data we gather. Some features are going to have a greater or lesser accuracy depending on the resources allocated to them. If we assume we have already gathered some *training* data, we can use it to make decisions about how to spend a budget in gathering future *test* data, assuming a budget may be split over features.

For example, in the design of a sensor network a budget is split over the locations: choosing where to place each sensor and the quality of sensor to be placed. Some sensors may be more expensive but more accurate. We refer to these locations as "contexts", indexed by c, and assume we have some initial training data $\{\mathbf{X}\}_{n=1}^{N}$. Contexts and training data are illustrated on the left in Figure 3.1 and the flow chart shows how an expected budget informs the learning procedure.

Initial training data may be from another sensor network or simulation. This training data is treated as noise-free inputs, allowing us to simulate the noisy test data $\tilde{\mathbf{X}}^n$ by assuming a distribution $p(\tilde{\mathbf{X}}^n|\mathbf{X}^n, \boldsymbol{\gamma})$, where $\boldsymbol{\gamma}$ parameterises the budget; defining the relationship between resources allocated and the noise on each context in $\tilde{\mathbf{X}}^n$. Figure 3.1 shows how this learnt budget and model is then used at test time to produce predictions.

In contrast to a traditional machine learning problem, this means we are assuming the training and test datasets are drawn from *different distributions*, but we can still express an objective in terms of the expected problem at test time. The problem of learning a relationship between $\tilde{\mathbf{X}}^n$ and $\mathbf{y}^n$ can be framed as an optimisation problem

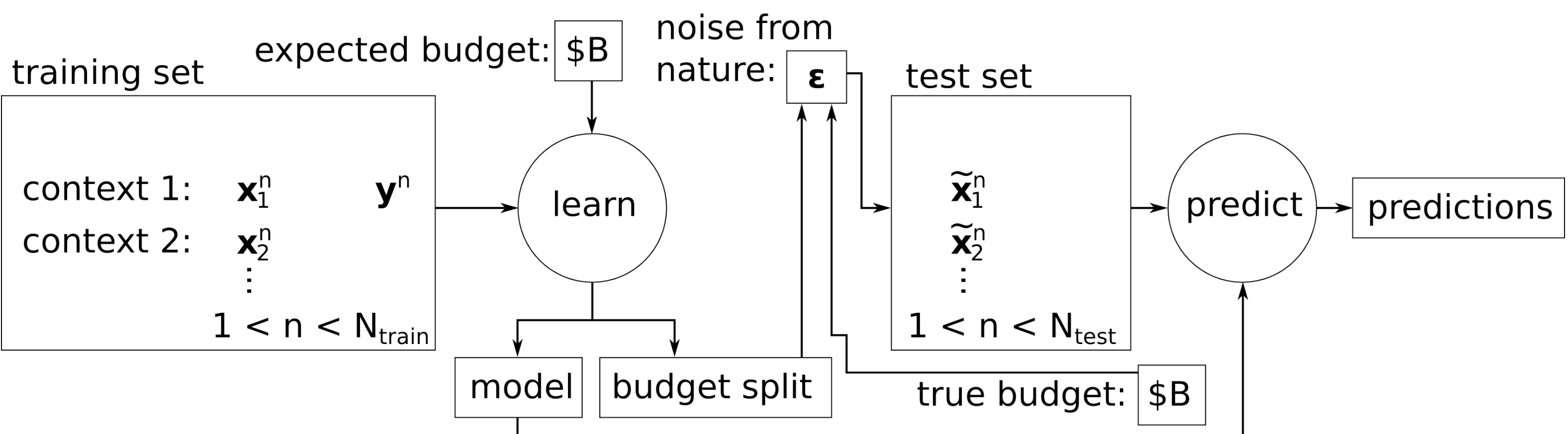


Figure 3.1: On the left the training set $\{\mathbf{X}^n\}_{n=1}^{N_{train}}$ is shown, illustrating the separation of features within each datapoint into contexts. These can be indexed by c: $\mathbf{x}_c^n{}_{n=1}^{N_{train}}$. In this learning process, a budget B is considered during training. Training then produces not just the trained *model*, but also a *budget split* to use at test time. The budget split then affects the noise incident on our trained model at test test time through the application of a real budget. The final result of this process are the predictions to be used in an application.

by empirical risk minimisation (Murphy, 2012, p.205). We are trying to find the model and budget split such that we minimize our expected loss:

$$\theta^*, \gamma^* = \arg\min_{\theta,\gamma} \mathbb{E}_{p(\tilde{\mathbf{X}},\mathbf{y}|\gamma)} \left[l(f_\theta(\tilde{\mathbf{X}}), \mathbf{y}) \right], \tag{3.1}$$

where our model is denoted f_θ, parameterised by θ, and l is a scenario dependent loss function.

In addition to a sensor network, another example setting could be in social science: we could survey in different locations or different categories of people (Lynch, 2007). The more people surveyed the more accurate the sample for that location/category. Or, in time-limited electromagnetic imaging we may wish to focus the scanning location on one region over another (Wintenby and Krishnamurthy, 2006).

Contexts are expected to contain different information; some may be informative for detecting one class over another. For classes where the data contains anomalies, it could be valuable to focus the budget on contexts containing more information on that class. This may allow us to mitigate the effect of anomalies on performance.

The conditional probabilistic model f_θ is assumed to be constructed as a differentiable function, and we find θ^* using gradient optimisation. The parameters defining these models are learnt by optimisation, usually by evaluating the likelihood over a dataset of observed examples.

In this chapter, we propose using the same gradient-based algorithms to learn a budget affecting the expected noise on the test set. The budget is included as an unconstrained parameter transformed to split a budget allocation using a softmax function. Using the reparameterisation trick (Kingma and Welling, 2014; Bonnet, 1964; Price, 1958; Salimans and Knowles, 2013) to sample noise variables, we can update the parameters of the budget directly as we learn our model. These gradient methods have the advantage of scaling to large or high-dimensional datasets easily.

In sum, this chapter establishes a new simultaneous method to anticipate a budget and produce a predictive algorithm. Further, we allow that the models used may be complex and nonlinear. Using these nonlinear models we show that it is possible to encode specific prior knowledge in applied problems and save resources in a way that would previously be impossible. We provide a recipe for designing a model in such settings and show how to apply stochastic gradient optimisation algorithms to produce useful models and budgets.

In Section 3.2 the details on how the noise process may be modeled is described. We explore the different types of noise processes we will consider in Section 3.2.1. Previous work on considering noise at test time is investigated in Section 3.2.2.

The benefits of designing our model to anticipate a budget are investigated by experiment in Section 3.3. We present a toy problem to demonstrate the effectiveness of this method on high-dimensional data in Section 3.3.1. A task of inferring digit rotation from sets of images of rotated digits is set, and a budget is learnt specifying which digits are more useful.

In Section 3.3.2 we focus on a real world sensor network imputation problem. We find that the method is effective at conserving resources assuming a price on the information coming from each sensor. Finally, in Section 3.3.3 we present a way to learn the quantisation matrices for natural image compression automatically from data. Due to the flexibility of the model, the effectiveness of simple gradient optimisers and increased computational resources, this is now a plausible method, and it is less costly than tuning such quantisation matrices manually.

3.2 METHODS

Requiring a budget to be specified at test time adds a decision problem to the task of minimizing the loss over an unseen test set. We address this by incorporating a model of how the budget will affect the expected loss at test time, as expressed in Equation 3.1. We find that this model of the budget is then efficiently optimised using stochastic gradient descent, given an appropriate parameterisation.

Our model could be conditional on features that are statistics obtained by *aggregating samples*, so the cost of a given accuracy is linear in the number of samples that need to be collected, for Normally distributed random variables, to achieve that accuracy. Alternatively, the model could be conditional on features that are quantised according to the budget allocation. An example of this is given in Section 3.3.3. The sampling variance is normally distributed if this is how our features are constructed. Even with non-Gaussian features, under weak conditions, statistics will be approximately Gaussian distributed due to a central limit theorem (Jaynes et al., 2003, p.222).

One form of central limit theorem states that if N IID random variables $\{X_i\}_{i=1}^N$ with mean μ and variance $\sigma^2 < \inf$ are combined to form $S_N = \sum_{i=1}^N X_i$, then as N approaches infinity the random variable S_N will converge in distribution to Normal (Murphy, 2012, p.51):

$$p(S_N) \approx \frac{1}{\sqrt{2\pi N\sigma^2}} \exp\left(-\frac{(s - N\mu)^2}{2N\sigma^2}\right) \tag{3.2}$$

In the case of the random variables, X_i, mentioned above it was already noted that their sum would have variance $N\sigma^2$. As the variance scales linearly with the random

variable, the mean $\sum_{i=1}^{N} X_i/N$ will have variance $N\left(\frac{\sigma^2}{N^2}\right) = \frac{\sigma^2}{N}$. This is the *standard error*, a noise term with standard deviation proportional to $\frac{1}{\sqrt{(N)}}$ but beginning (at $N = 1$) with an unknown variance of σ^2.

In the following section we discuss how to incorporate this observation about the variance of aggregated samples into a probabilistic model. We define how this model is constructed functionally, and how it may be learnt using stochastic gradient optimisation.

3.2.1 *Noise Generating Processes*

The noisy inputs $\tilde{\mathbf{X}}^n$, see Equation 3.1, encountered at test time depends on the application. These varied noise generating processes are modeled by conditional distributions $p(\tilde{\mathbf{X}}^n|\mathbf{X}^n, \boldsymbol{\gamma})$. As mentioned before, a simple case would be when we are paying for each sample and taking an average to generate a feature. From the above definition of standard error, we have:

$$p(\tilde{\mathbf{X}}^n|\mathbf{X}^n, \boldsymbol{\gamma}) = \mathcal{N}\left(\tilde{\mathbf{X}}^n; \mathbf{X}^n, \mathrm{diag}(\boldsymbol{\sigma}(\boldsymbol{\gamma}))\right), \tag{3.3}$$

where $\boldsymbol{\sigma} = \sigma_c{}_{c=1}^{C}$ depends on the resources r_c allocated on each context c, as the number of samples we have in each context is directly proportional to the resources, e.g. number of samples, allocated:

$$\sigma_c = \frac{1}{\sqrt{r_c}}. \tag{3.4}$$

This is the standard error, and relevant in many scenarios due to the relationship with the central limit theorem in Equation 3.2.

Richman and Mannor (2016) details two other cases. First, if the resource is a sampling rate and features are the timing of events then our noise standard deviation would be exactly inversely proportional to the resources allocated:

$$\sigma_c = \frac{1}{r_c} \tag{3.5}$$

Second, quantisation noise is often approximated as a Gaussian noise source, as we do in Section 3.3.3. If we are paying for a given bit resolution then the noise standard deviation can be the following:

$$\sigma_c = 2^{-r_c} \tag{3.6}$$

From these definitions, if a linear model is assumed it is possible to define closed form expressions for the resources to allocate to each feature, given a total budget $R = \sum_i r_i$ (Richman and Mannor, 2016). We focus on nonlinear models so have to resort to iterative optimisation algorithms. These algorithms are run to find a useful setting of the parameters θ and budget split γ, according to the minimisation of a loss l, as described in Equation 3.1. We choose to minimize the negative log likelihood (NLL) of our model given the data.

At training time we expect to have a supervised problem defined by a dataset $\mathcal{D} = \{\mathbf{X}^n, \mathbf{y}^n\}_{n=1}^N$. We assume that the test set will differ from the training set. This noise will depend on the budget allocation, parameterised by γ. After adding noise we have a variable $\tilde{\mathbf{X}}^n$, leading to the belief network shown in Figure 3.2.

Here, we assume that the function f_θ used in our model, as described in Equation 3.1, is a nonlinear and differentiable. It is a function defining the conditional model $p(\mathbf{y}^n|\tilde{\mathbf{X}}^n;\theta)$, parameterised by θ. The relationship between $\mathbf{X}^n$ and $\tilde{\mathbf{X}}^n$ is incorporated through a differentiable noising function g_γ, parameterised by γ,

$$\tilde{\mathbf{X}}^n = g_\gamma(\mathbf{X}^n, \epsilon), \tag{3.7}$$

where ϵ is a random variable that we can sample easily. For example, in our experiments we use the common reparameterisation of a conditional Gaussian with mean $\mathbf{X}$ and variance $\sigma(\gamma)^2$:

$$g_\gamma(\mathbf{X}^n, \epsilon) = \mathbf{X}^n + \sigma(\gamma) \odot \epsilon. \tag{3.8}$$

In this case ϵ is a standard Gaussian with mean 0 and variance 1.

To optimise θ and γ we need to evaluate the expected loss in Equation 3.1, and differentiate it with respect to θ and γ. This can be achieved by taking L samples of ϵ (Kingma and Welling, 2014):

$$\mathbb{E}_{p(\tilde{\mathbf{X}}^n, \mathbf{y}^n|\gamma)}\left[l(f_\theta(\tilde{\mathbf{X}}^n), \mathbf{y}^n)\right] = \mathbb{E}_{p(\tilde{\mathbf{X}}^n, \mathbf{y}^n|\gamma}\left[l(f_\theta(g_\gamma(\mathbf{X}^n, \epsilon)), \mathbf{y}^n)\right] \tag{3.9}$$

$$\simeq \frac{1}{L}\sum_{l=1}^{L} l(f_\theta(g_\gamma(\mathbf{X}^n, \epsilon)), \mathbf{y}^n), \tag{3.10}$$

where our loss function l is also differentiable, we can evaluate gradients with respect to θ and γ.

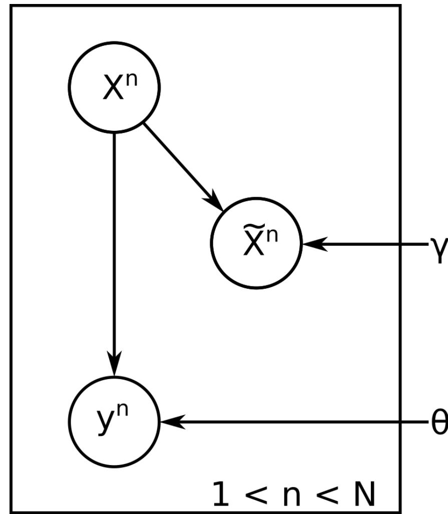


Figure 3.2: A supervised task with noisy inputs $\tilde{\mathbf{X}}^n$, noise-free true inputs $\mathbf{X}^n$ and targets $\mathbf{y}^n$. At training time we observe $\mathbf{X}^n$ and $\mathbf{y}^n$, while at test time we have only $\tilde{\mathbf{X}}^n$.

To optimise with respect to $\boldsymbol{\gamma}$, we need to define $\boldsymbol{\sigma}(\boldsymbol{\gamma})$, as we have defined $g_{\boldsymbol{\gamma}}$ in terms of $\boldsymbol{\sigma}(\boldsymbol{\gamma})$. In the case where we are gathering samples and learning from statistics, the appropriate relationship between the resources allocated and $\boldsymbol{\sigma}$ is shown in Equation 3.4. To split the budget, an unconstrained $\boldsymbol{\gamma}$ variable is passed through a softmax function:

$$\sigma_i^2 = \frac{\nu_i}{B \times \text{softmax}_i(\boldsymbol{\gamma})}, \tag{3.11}$$

where ν_i is a present original variance.

Alternatively, we could be allocating a budget of a number of bits in a quantisation scenario, such as that described in Equation 3.6. In this case, the softmax function is used to divide B bits between contexts:

$$\sigma_i^2 = 2^{-2B \times \text{softmax}_i(\boldsymbol{\gamma})} \tag{3.12}$$

Using a differentiable model, and sampling from simple noise distributions, we are able to build complex nonlinear models and learn the parameters of such models on large datasets. The training algorithm expresses our assumptions about the noise seen at test, allowing us to minimise the loss on an unseen test set matching those assumptions. We are free to use popular stochastic gradient algorithms common in deep neural network research. As such, this is a flexible, scalable method for incorporating budgets into probabilistic models.

3.2.2 *Related Work*

Richman and Mannor (2016) provides the closest basis for the work presented here, the difference being that they focus on linear models while we focus on nonlinear models. They find that in the case of linear models, for all three of the cases described in Equations 3.4, 3.5 and 3.6, it is possible to derive a closed form expression for a budget split. Other relevant work in the field does not necessarily produce a budget split, but focuses instead on taking into account the cost of features.

Optimising an explicit objective function while minimising the test time cost involves some penalty term associated with "expensive" features. This is typically framed as a sequential problem, independently evaluated over examples. Starting with a given feature, the algorithm can predict or decide which new features to choose. The algorithm may therefore be able to incorporate different features on each example at test time. Our method spends the same budget on every example but divides resources among contexts according to a budget split that is shared over all examples.

Work in this area has focused on parametric models such as Decision Trees (Nan and Saligrama, 2017), Random Forests (Nan et al., 2015, 2016) or boosting (Peter et al., 2017; Xu et al., 2012). These models typically come up with a heuristic to select features at test time, along with a trained model that can operate on a varied set of input features.

Or, a more generic method could be to treat this as an explicit sequential decision problem and apply reinforcement learning, such as Q-learning (Janisch et al., 2017; Contardo et al., 2016) or structured prediction (Bojarski et al., 2016). In either case, we cannot evaluate gradients for the problem of selecting a sequence of features on each example, so these gradient-free algorithms are providing a possible way to learn this strategy.

Optimisation without gradient information progresses much more slowly as the number of dimensions in a problem increases. In addition, trying to compose a larger machine learning system, and learn the parameters of such a system, using a gradient-free method will be more difficult still. Additional components add to the dimensionality of the problem and increase the time needed to find a solution. In contrast, a gradient optimised model can incorporate data or other structures into the computational graph due to its generic nature as a stochastic computation graph (Schulman et al., 2015).

Our stochastic computation graph describes a conditional model, which we are able to optimize through sampling. An example of an unsupervised model learnt in

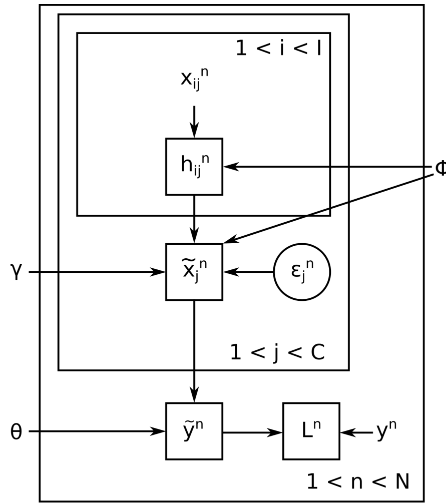


Figure 3.3: A stochastic computation graph (Schulman et al., 2015) describing the inference algorithm applied to the model described in Figure 3.2. Inputs x_{ij}^n at training time are optionally processed by a statistic network, parameterised by $\boldsymbol{\phi}$, to create a representation h_{ij}^n. Taking the mean of this representation produces a statistic used for classification, which passes through a noising function, as described in Equation 3.7, to produce $\tilde{\mathbf{x}}_j^n$. This is then used to produce a prediction $\tilde{\mathbf{y}}^n$ which, along with a target $\mathbf{y}^n$, produces a scalar loss L^n. As every step in this graph is differentiable, we are able to then optimise θ, $\boldsymbol{\phi}$ and $\boldsymbol{\gamma}$ with respect to this loss.

a similar way would be a Variational Autoencoder. We can contrast Equation 3.10 with the variational ELBO (Kingma and Welling, 2014) (evidence lower bound):

$$\tilde{\mathcal{L}}^B(\theta, \boldsymbol{\gamma}; x^{(i)}) = -D_{KL}(q_{\boldsymbol{\gamma}}(\mathbf{z}|\mathbf{x}^{(i)})||p_\theta(\mathbf{z})) + \frac{1}{L}\sum_{l=1}^{L} \log p_\theta(\mathbf{x}^{(i)}|\mathbf{z}^{(i,l)}) \tag{3.13}$$

D_{KL} here refers to a KL divergence, here between the approximate posterior and the prior, and $\log p_\theta$ is the log-likelihood of the model given the data. For example, the log-likelihood may be a squared error on real-valued data. The choice of prior decides the regularizing effect of the KL divergence in this loss function. A prior with large support may allow the autoencoder to pass information through while adding little noise. In our algorithm, we have no regulariser, but the softmax parameterisations described in Equation 3.11 and 3.12 require a commensurate increase in noise elsewhere when noise on any context is reduced.

3.2.3 *Simultaneous Statistic Optimisation*

Figure 3.3 provides a description of how the learning system operates as a stochastic computation graph (Schulman et al., 2015). Square elements represent deterministic nodes, round elements are stochastic nodes and the free nodes are inputs. The graph

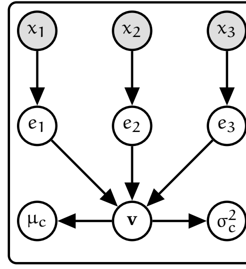


Figure 3.4: An example statistic network, gathering observations x_1, x_2 and x_3 to make a statistic in the representation learnt by the network (Edwards and Storkey, 2017).

illustrates the incorporation of noise $\boldsymbol{\epsilon}$ through parametric noising function g_{γ} and how the resulting $\tilde{\mathbf{x}}_j^n$ is used to create a prediction $\tilde{\mathbf{y}}^n$. With a target $\mathbf{y}^n$ the loss function produces a scalar loss L, which can be differentiated with respected to $\boldsymbol{\theta}$, $\boldsymbol{\phi}$ and $\boldsymbol{\gamma}$.

In addition to the system described in previous section we include a node $\mathbf{h}^n$ from which statistics are computed. To account for this, we must modify Equation 3.7 to be:

$$\tilde{x}_j^n = g_{\gamma}\left(\frac{1}{I}\sum_{i=1}^{I} h_{\phi}(x_{i,j}^n), \epsilon_j\right), \tag{3.14}$$

where h_{ϕ} is a function that computes a representation from which the sample mean can be used to produce $\tilde{\mathbf{x}}^n$, and this is called a *statistic network* (Edwards and Storkey, 2017).

This abstraction is relevant because in applications we may be gathering multiple samples at each location to then combine into a statistic that can be used for the overall prediction problem. In this case, the network can incorporate this process and construct a representation to compute the most useful statistics.

Optimising the statistic calculated by a neural architecture involves propagating multiple examples through the same network, then averaging the vector at some hidden point. It can be used to infer a latent variable over a collection of examples and is used in the Neural Statistician to create a generative model over datasets (Edwards and Storkey, 2017). Alternatively, a statistic network can be used to learn a conditional model, as we do here. Other examples of such conditional models are prototypical networks (Snell et al., 2017) or neural processes (Garnelo et al., 2018).

An example statistic network is illustrated in Figure 3.4. The input variables are all passed through a network to produce hidden representations e_1, e_2 and e_3. These are then gathered into the statistic ν, which is used to produce the moments of the noise distribution μ_c and σ_c^2.

At training time, we gather the statistic from a *fixed* number of examples. At test time, we gather a variable number of samples to make that same statistic. By adding noise to the statistic, we seek to emulate the noise on that statistic from having fewer samples, but it is not precisely the same process. In Section 3.3.1 we show that in practice the variance due to fewer samples is effectively emulated by the addition of noise at training time.

In the following section we will detail a toy problem where the learning of a γ budget parameters along with the θ and ϕ parameters of the predictive model, and statistic network, is key to the solution. This solution is then compared to other candidate solutions, and we find that the budget split produced in a single optimisation matching those found after many iterations of a Bayesian Optimisation search.

3.3 EXPERIMENTS

Experiments in Sections 3.3.1 and 3.3.2 were run using Theano (Al-Rfou et al., 2016) and Lasagne (Dieleman et al., 2015), with GPyOpt (The GPyOpt authors, 2016) for Bayesian Optimisation. All figures were produced using Holoviews (Stevens et al., 2015). The code implementing all experiments is publicly available[1].

3.3.1 *Digit Rotation Inference*

We introduce rotational MNIST as a synthetic problem of inferring the rotation angle of a set of MNIST digits (LeCun et al., 1998) to demonstrate functionality on high-dimensional complex data. An input/output example is illustrated in Figure 3.5: on the left is the input to the network, a set of images where each context is the number in each image, and on the right is the target, the angle every image has been rotated by. This dataset is used to illustrate the optimisation of the statistic gathering procedure, test time performance, and robustness to required budget variation.

The following experiments on rotational MNIST were performed with an induced sparsity on the MNIST images. According to a probability associated with each context an image was randomly zeroed, which induces sparsity over the input examples. Otherwise, the task was typically solved without learning any budget other than uniform.

Examples are input to a *statistic network* as described in Section 3.2.3, and our learning algorithm operates as described in Section 3.2.1. The statistic network h_ϕ

[1] Implementations of experiments in Section 3.3.1 and 3.3.2 can be found at `https://github.com/BayesWatch/context-budget`. Implementations of the experiments in Section 3.3.3 use PyTorch (Paszke et al., 2017) can be found at `https://github.com/BayesWatch/bit-budget`.

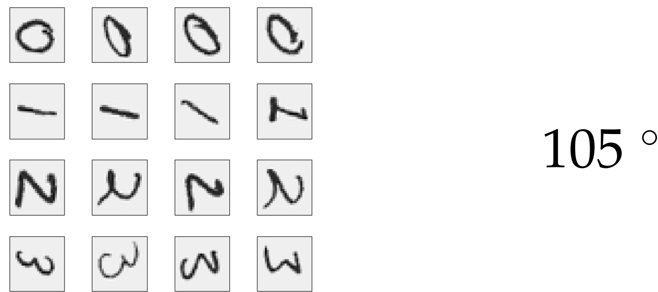


Figure 3.5: Rotational MNIST input (left) and output (right). Each context is a digit class, and at test time the budget decides how many images from each class we receive.

and predictive model f_θ were implemented as deep neural networks. At test time the statistic is gathered from the prescribed number of images given by the budget.

To contrast with our gradient-based optimisation method, we also tried a more computationally intensive black box optimisation method called Bayesian optimisation. Bayesian optimisation sets a prior function relating the budget to the expected performance and then updates that function on the evidence it receives from repeated attempts. The algorithm is able to trade off exploration and exploitation as it searches the space, although this depends on the chosen acquisition function (Brochu et al., 2010). GPyOpt (The GPyOpt authors, 2016) was used to optimise a budget prescription, training a network from scratch on each trial.

In total, we compared the performance of a budget found using our method to that of three other methods:

- Bayesian optimisation
- A uniform budget assigned to all contexts
- Using a model learnt by a uniform budget, we take the L2 norm of the weights in the layer combining the statistics from each context

In this last context, the L2 norm will provide an estimate of importance as it does in recursive feature elimination (Guyon et al., 2002) and is therefore a reasonable heuristic against which to compare.

The histogram on the right of Figure 3.6, illustrates the trials executed during a Bayesian optimisation search (The GPyOpt authors, 2016) over possible budgets, while red vertical lines illustrate the performance of the uniform budget and the budget based on the L2 norm. Our gradient optimised method achieves an average rotational error of 0.156 radians, which is marginally worse than 0.148: the error achieved using all of the Bayesian optimisation trials illustrated. Both of these perform much better than the other two competing methods, while our method is necessarily faster

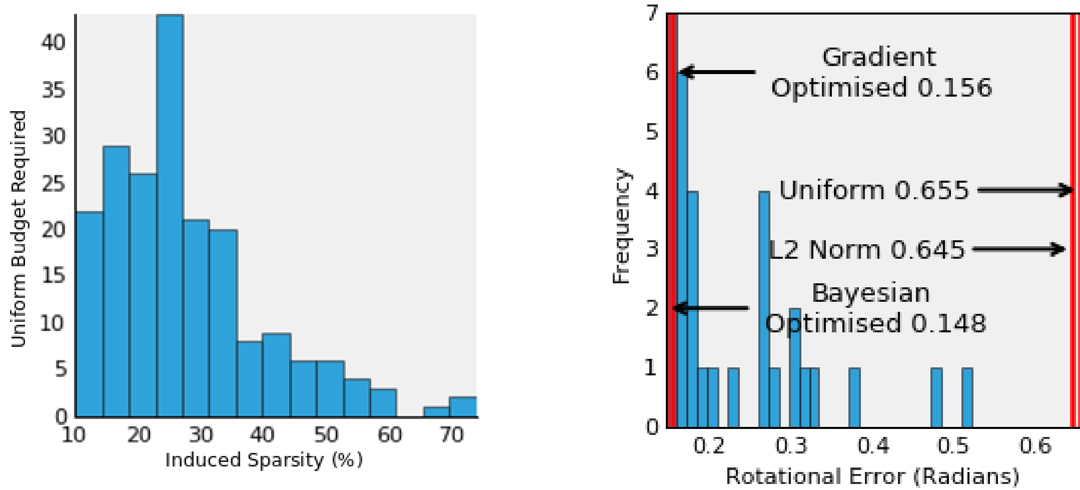


Figure 3.6: Left: comparing over different sparsity settings and a total budget of 10, the distribution of total budget required when uniformly split to match the performance of a gradient optimised allocation. Right: histogram of Bayesian optimised budget splits with vertical lines for the competing methods considered.

than Bayesian optimisation as optimising the budget parameters, γ, happens simultaneously as we learn the model parameters in a *single trial*.

The effect of induced sparsity is explored in the left histogram of Figure 3.6. As the probability of dropping an example is increased, the size of a uniform budget required to perform equal to the gradient-optimised budget decreases. At high levels of induced sparsity, it is cheaper to predict without any information.

For example, for a total budget of 10 split using our algorithm and an induced sparsity of 25%, a 4 times larger uniform budget was required in order to match performance. Over the variation in induced sparsity, we found that on average a uniform budget would have to be 173.3% +/- 12.6% greater to obtain the same performance as an optimised budget split.

In general, our approach was able to learn effective budgets in this toy problem. Due to the small size of the problem, Bayesian optimisation was also able find an effective budget split. It was allowed to run a large number of experiments due to the relatively small size of the problem. If each trial were more costly, then it would not be practical to run a large number of trials for the convergence of a Bayesian optimisation routine. In cases where experiments are costly, being able to obtain results in one training run using our method would be preferable to Bayesian Optimisation.

Implementation Details The statistic network was implemented as a Multi-Layer Perceptron (MLP) with two hidden layers, 800 units each, with batchnorm and dropout set to 0.5 during training, producing 32 statistics. After g_γ these were passed to an MLP with a single hidden layer. Rotational regression was performed by discretising

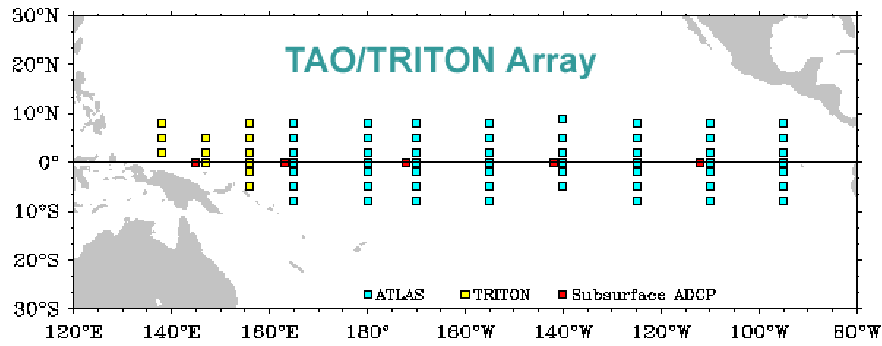


Figure 3.7: Illustration of the Tropical Atmospheric Ocean (TAO) array sensors for monitoring the El Niño event (Lichman, 2013).

the space of radians into 256 classes and treating the problem as classification. Optimisation was performed using Adam (Kingma and Ba, 2014) with learning rate 0.001, $\beta_1 = 0.9$, $\beta_2 = 0.999$ and $\epsilon = 1 \times 10^{-8}$.

3.3.2 *Sensor Array Imputation*

For this experiment, we use the real data from the Tropical Atmospheric Ocean (TAO) sensor array, which focuses on the El Niño event (Lichman, 2013) and is composed of around 70 sensor buoys in the Pacific Ocean. We try to infer the observations at each buoy given the observations at all other buoys, attempting to quantify relative informativeness of each individual buoy. The placement of the buoys is illustrated in Figure 3.7. Over 50 Buoys are placed on a rough grid over a large area of the Pacific ocean, between 135°E and 95°W. To maximise the number of temporally concurrent observations, a subset of 10 buoys were selected from the dataset over the whole array. This subset is illustrated in Figure 3.8, showing the relative locations of these 10 buoys using normalised measurements of latitude and longitude that were used in processing.

We assume that the resources allocated to each buoy will reduce the noise at each location according to Equation 3.11. A small MLP was used to impute the observations at each buoy, using the observations at all other locations, as a regression problem.

On the 10 different imputation tasks, we found the uniform budget allocation that would match the performance of our gradient-optimised budget split and evaluated the uniform extra cost as the difference between total budgets in each allocation setting. The results of this experiment are illustrated in Figure 3.9, showing that some buoys require far more resources using a uniform budget versus one found by gradi-

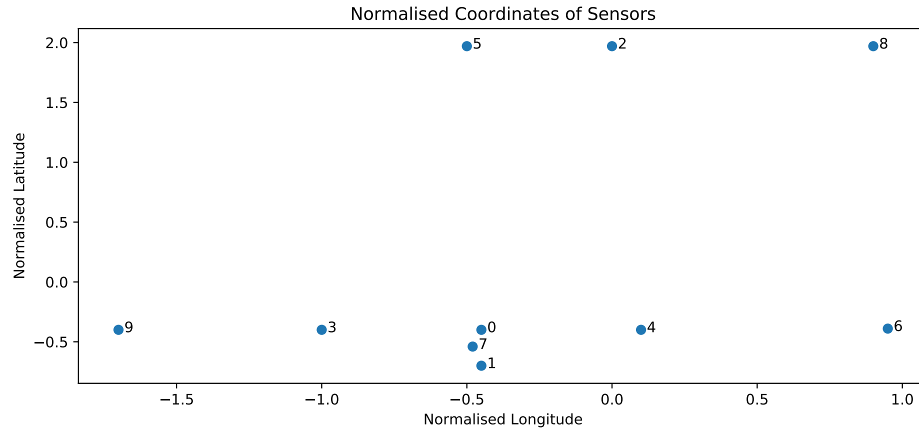


Figure 3.8: Normalised sensor placement of the subset of sensors used in the experiment on the TAO dataset described in Section 3.3.2.

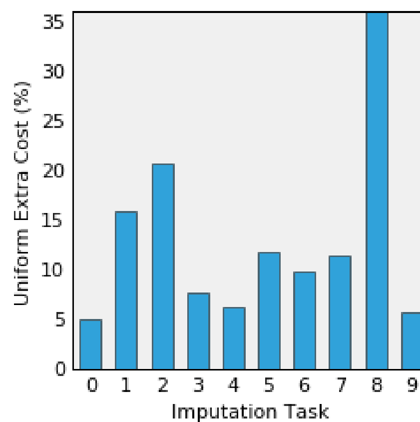


Figure 3.9: The extra cost of using a uniform budget allocation over one found by gradient optimisation. Each imputation task is the problem of inferring the observations at any buoy. Buoys 2 and 8 are from proximal sensors, and learn budgets preferring data from each other.

ent optimisation. We find that overall, averaged over all imputation tasks, the gradient optimised budget could be 13% smaller.

However, on certain imputation problems we can see a much greater benefit. In Figure 3.9 columns 2 and 8, imputing observations at buoys 2 and 8, show a uniform budget split may cost 20% to 35% more. This could be attributed to the location of these buoys. Figure 3.8 shows that these two buoys are close together, and in a group of three far from the majority of the other sensors. In Figure 3.10 we plot the budget splits learnt by each of these two buoys and see that each of these imputation tasks does favour information from the most proximal buoy, as we would expect.

We may then also expect to see a similar cost saving over a uniform budget for buoy 5; in Figure 3.8 it is in the same group as buoys 2 and 8. One explanation we can give for this is that the dataset is noisy and has many missing features. Given the

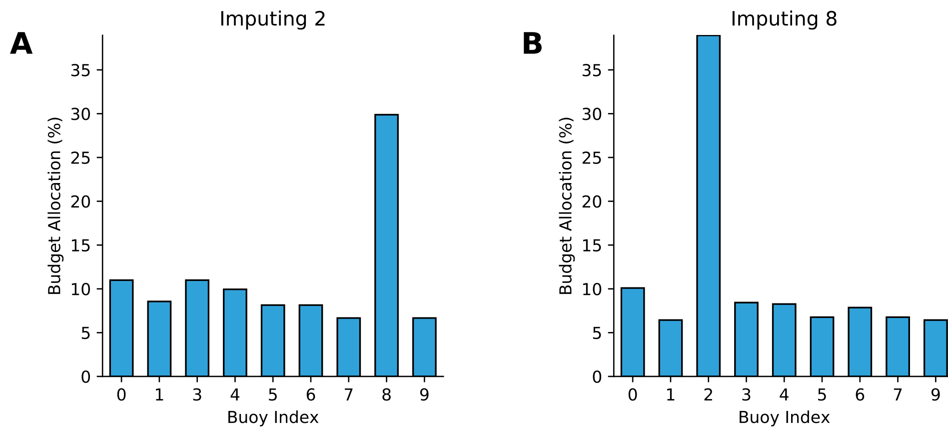


Figure 3.10: Illustrating the budget splits learnt when imputing the observations at buoy 2 and buoy 8 on the TAO sensor network imputation experiment investigated in Section 3.3.2. In both cases, the most proximal buoy, as shown in Figure 3.8 is favoured over all others.

noise, buoy 5 could be imputed more easily by predicting the mean of each feature. The normalised mean squared error when predicting the mean for buoys 2 and 8 was 2.88 and 2.697, respectively, in contrast to buoy 5, where predicting the mean achieved 0.83.

Real world datasets present problems that we do not see in, for example, the toy problem investigated in Section 3.3.1. In this section we have demonstrated that we can learn useful budget splits despite these difficulties, and we have observed that the allocation of resources makes sense in the context of the problem.

Implementation Details The original dataset was reduced to only 10 sensors. Observations were grouped to local contexts using DBSCAN (Ester et al., 1996), a clustering method, and visual inspection. The subset of 10 with the largest set of contiguous observations was then chosen greedily, starting with the location with the largest number of observations.

In this subset, 8.3% of all features were found to be missing. These were replaced with the mean value for that given feature, in that context. In addition, a dimension was supplied to the regression model indicating which features were missing, and this was also used to remove such features from the loss calculation. The mean squared error did not include contributions from features that had been set to that feature's mean value.

The MLP used for imputation had 3 hidden layers and used 96 hidden units each. Between layers batch-norm was applied.

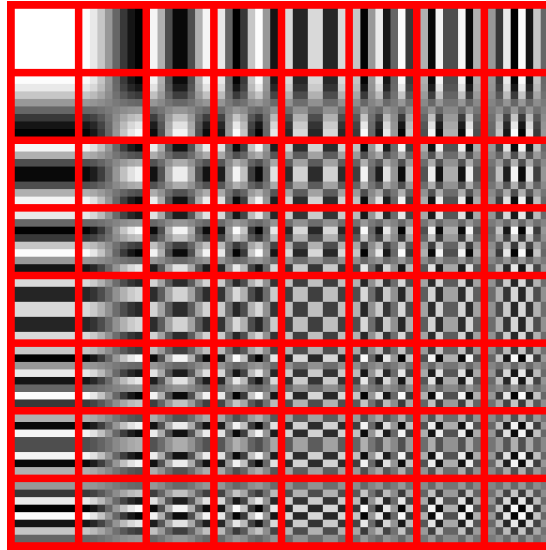


Figure 3.11: The 2D discrete cosine transform decomposes an 8 by 8 patch using a linear combination of these patterns (Wikimedia, 2019). These are referred to as the 2D DCT basis functions.

3.3.3 *Learning Image Quantisation Matrices*

A common method for lossy compression of images is the JPEG algorithm. This algorithm relies on hard-coded quantisation matrices, which allow a user to trade off perceptual image quality against its stored size. In this experiment, we use our method to learn quantisation matrices that achieve the same goal directly from the image. The budget in this case reflects the stored size acceptable to the user.

The JPEG algorithm applies a 2D discrete cosine transform (DCT) to the image on every individual 8 by 8 pixel patch in the image. Each patch is then represented as a linear weighting of the 2D DCT basis functions illustrated in Figure 3.11. After producing these coefficients, it then attempts to quantize them – in other words, reduce their bit depth – in such a way as to preserve the quality of the image according to human perception. This involves applying a quantisation matrix, followed by lossless compression methods: typically run-length encoding followed by Huffman coding (Wallace, 1991).

It should be noted that this means that when we learn how many bits to allocate to each DCT coefficient out of the total budget, the size of the resulting stored file is only *correlated* to that total budget. The process by which lossless compression algorithms operate is non-differentiable. However, it is possible to estimate the expected number of bits prescribed by a given quantisation matrix if the implied bit depth was applied instead. This is discussed in more detail after we define quantisation matrices below.

The Discrete Cosine Transform The DCT is a real-valued analogue of the discrete Fourier transform. There are four types of DCT and in this work we refer to the DCT-II as the DCT, respecting the conventional use (Ahmed et al., 1974). In 1D, the DCT transforms a vector, $\mathbf{x}$, into the appropriate coefficients, X_k, to weight a set of cosine functions of different frequencies in order to reproduce said vector:

$$X_k = \sum_{n=0}^{N-1} x_n \cos\left[\frac{\pi}{N}\left(n + \frac{1}{2}\right)\right]. \tag{3.15}$$

Where $k \in (0, N-1)$. In 2D, this transformation is analogous but first applied to the rows and then the columns of the 2D input.

Quantisation Matrices A quantisation matrix $\mathbf{Q}$ is applied by element-wise division of the incoming DCT coefficients and rounding the result (Wallace, 1991):

$$z_k = \text{round}\left(\frac{X_k}{Q_k}\right) \tag{3.16}$$

The DCT coefficients may have a greater bit depth than the integers defining the image pixels, which are expected to be 8 bit. However, in our work, as an approximation we assume the input signal and DCT coefficients to be initially quantised to 8 bits.

In this experiment, we denote the maximum bit depth as $\beta = 8$ in order to define $\mathbf{Q}$ in terms of the budget split $\mathbf{B}$:

$$Q_{ij} = 2^{\beta - B_{ij}}, \tag{3.17}$$

where $\beta - B_{ij}$ is the implied bit depth of the DCT coefficient at index i, j.

We parameterise our budget split by the unconstrained budget parameters $\boldsymbol{\gamma}$ and the total budget B_{total} using the softmax function:

$$B_{ij} = B_{\text{total}} \times \text{softmax}_{ij}(\boldsymbol{\gamma}). \tag{3.18}$$

This is a relaxation of the discrete values required for quantisation to an exact bit depth.

Once we have calculated the DCT coefficients for any 8 by 8 patch, the inverse 2D DCT exactly reproduces the original patch. Adding noise prior to the inverse transform is a way to approximate the effects of quantisation. After reproducing the image with this type of quantisation applied, we have to decide which metric to use to express the perceptual error due to quantisation.

Perceptual Error Using simple squared error is not sufficient because the human eye does not respond to all frequency components in natural images equally (Wallace, 1991). We can use a learnt model of perceptual error by backpropagating through a model parameterised by a neural network.

The model we use in this experiment is the PieApp network (Prashnani et al., 2018). This network is a convolutional architecture trained on a dataset pairing images with a score given by a set of human subjects. Subjects were asked which of a set of images they prefer. After training, the network output is an approximation to the probability that a subject would prefer a given image. In our experiments, this provides both a practical way to score the final images produced after compression and a loss function we may use to train networks; allowing us to optimise towards images that have a high probability of being preferred by subjects.

Inverse Cumulative Density Function (CDF) Transforms The DCT coefficients may not be normally distributed and as such adding normally distributed noise may not modulate the signal to noise ratio. As noted in Section 3.2.1, the noise standard deviation is directly related to the bit depth used for a given DCT coefficient. In order to mimic this quantisation noise, both signal and noise have to be normally distributed. To enforce this, we use a piecewise linear CDF transformation to cast each dimension of the DCT coefficients into a normal space. After addition of a normally distributed noise in this space, we can then easily invert the piecewise linear transformation to return to the original space.

This is effectively a chained inverse transform sampling procedure. Given a uniform random variable, it is possible to transform samples from this random variable to any 1D distribution with a defined CDF function F (Devroye, 1986, p.31):

$$\Pr(F^{-1}(U) \leqslant x) = \Pr(U \leqslant F(x)) = F(x). \tag{3.19}$$

Decoding Using An MLP However, learning using a perceptual error network to propagate error gradients is very expensive. We also examine an alternative cheaper method to parameterise the transformation from the DCT space to image space using an MLP and then evaluating a squared error in pixel space. If the MLP is able to best use the available information, the budget learnt would be one that preserves the maximum information about the image being processed. We show by experiment that the flexibility of the MLP model is sufficient to obtain performance comparable to the perceptual error network.

Perceptual Comparison Once we have learnt quantisation matrices by either method, we apply them precisely as they are applied in the JPEG algorithm and use the resulting quantised representation to reproduce the image patches. To compare the resulting images, we again use the PieApp network (Prashnani et al., 2018) to give a preference of the learnt matrices over the original JPEG quantisation matrices, for different quality values used by the JPEG algorithm.

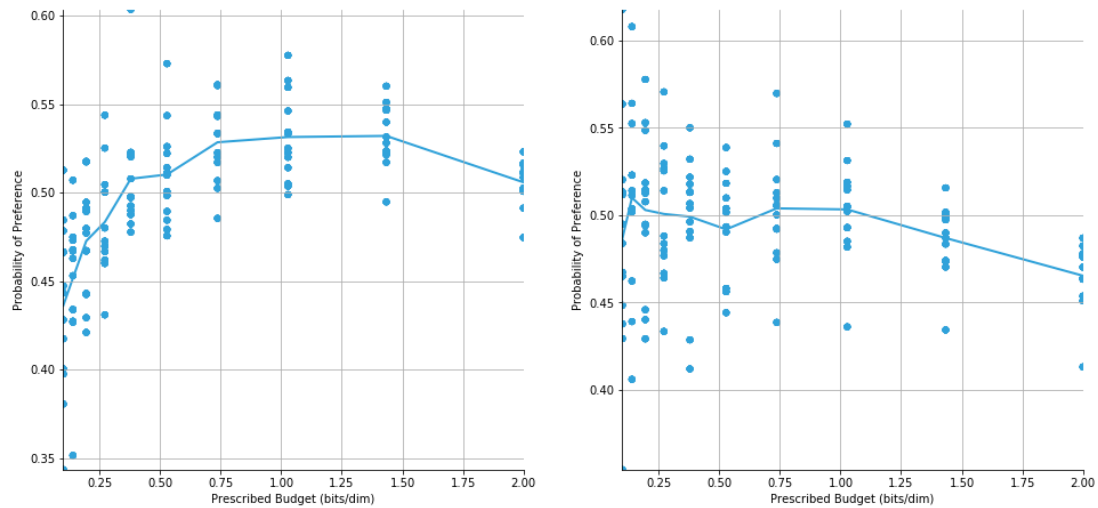
In Figure 3.12 we show how this preference varies depending on the budget (bits per pixel) prescribed. The left graph corresponds to the quantisation matrices learnt with the perceptual error network, while the right graph shows the matrices learnt with an MLP.

As noted above, the stored size of an image depends on lossless encoding algorithms, which means that the exact file size cannot be prescribed. However, in order to meaningfully compare the perceptual quality of images obtained with different quantisation matrices, the file sizes must be equal. To estimate the perceptual preference between two images of different sizes, we compare images produced by learnt quantisation matrices to many images produced over the range of JPEG quality values (and thus, of different file size). Using these values, we use linear regression to find the expected perceptual preference were the file sizes equal. These make up the scatter plots in Figure 3.12 for various stored image sizes. Figure 3.12a shows that images stored using quantisation matrices optimised for perceptual preference consistently score higher on that same perceptual metric. Figure 3.12b demonstrates that quantisation matrices produced *without* reference to the perceptual error metric, instead using an MLP for decoding, are equally preferred to JPEG compressed images.

The learnt quantisation matrices are preferred for images of stored size when using smaller budgets. At higher levels of compression the perceptual error likely does not change much depending on the budget split. All images will be perceived as approximately the same.

The results demonstrate that quantisation matrices can indeed be learnt automatically from data. The resulting matrices are interchangeable with respect to perceptual preference than the JPEG quantisation matrices and can be tuned to the individual image being stored. We further show that, by using an MLP to decode DCT coefficients, this learning process can be performed quickly, efficiently and *without domain knowledge*. This factor could be relevant for the compression of images in other domains, such as medical images or multispectral imaging.

We show this further in Figure 3.13, where we plot the probability of preference of the image obtained with perceptual error optimisation over that using an MLP to decode from the DCT space. Although we might not expect it, the MLP quantisa-



(a) Quantisation matrices learnt by optimisation of PieApp perceptual preference (Prashnani et al., 2018).

(b) Quantisation matrices learnt by decoding DCT coefficients using an MLP.

Figure 3.12: Learnt quantisation matrices are compared to the quantisation matrices prescribed by JPEG; "Probability of Preference" over a JPEG encoding of the same image. The PieApp pretrained perceptual error network (Prashnani et al., 2018) is used to give a probability of preference of the learnt matrix. Linear regression was used to estimate the perceptual preference if the file size were equal and these values are plotted against the prescribed budget.

tion matrices are preferred for small budgets but this is most likely to do with the difficulties with backpropagation through the perceptual error network.

The images learnt in either case look similar; Figure 3.14 compares quantisation matrices learnt by perceptual error, by using MLP, and the JPEG prescription, for a comparable file size. We find that, by visual inspection, the learnt quantisation matrices produce an image with fewer artefacts.

Discussion In this section we have presented two methods to learn JPEG quantisation matrices automatically from data. Using a published architecture as an approximation to perceptual quality (Prashnani et al., 2018), the probability a subject would prefer a given image, we were able to compare the performance of different quantisation matrices for compression. Both were successful in learning matrices that performed as well, or better than the hard-coded quantisation matrices applied in the JPEG specification. There are benefits to a learnt quantisation matrix: we were able to learn them from *single images* and, when using an MLP to decode DCT coefficients, entirely without domain knowledge. Being able to tune our compression method by explicitly defining a budget and apply it on other types of image data, such as medical or multispectral imaging, could be valuable in future work.

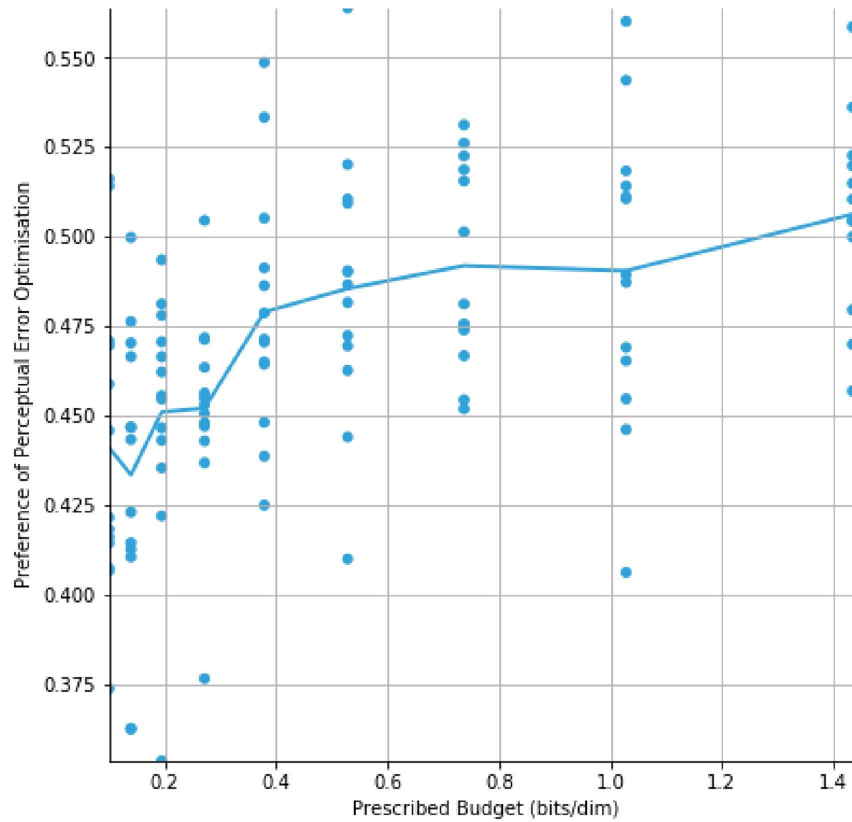
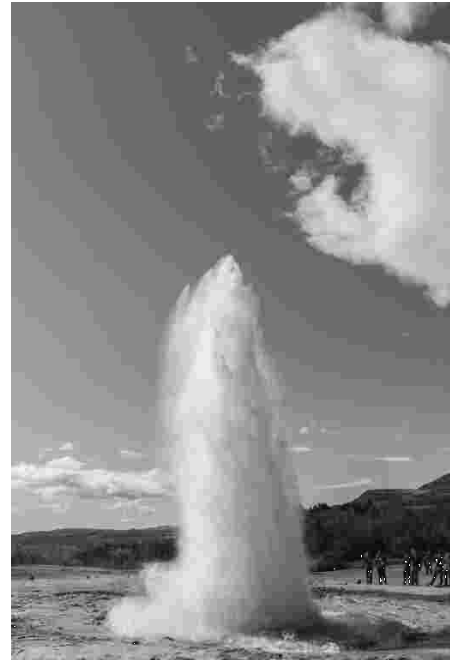


Figure 3.13: Comparing the perceptual preference of learnt quantisation matrices when they are learnt either by optimising perceptual error directly, or implicitly using an MLP. Different points correspond to different images.

(a) The original image, uncompressed stored size: 3145728 bits



(b) Quantisation matrix with a JPEG quality value of 10, stored size: 62241 bits.

(c) Quantisation matrix learnt by backpropagating perceptual error (Prashnani et al., 2018), stored size: 59465 bits.

(d) Quantisation matrix learnt using an MLP to decode the DCT coefficients and minimising the L2 distance in pixel space. Stored size: 70281 bits.

Figure 3.14: Example images created using quantisation matrices that are stored at comparable file sizes (prescribing an exact file size is not possible due to unpredictable lossless encoding).

Implementation Details The MLP used in decoding had a single hidden layer with 128 hidden units and batch-norm between layers. Optimisation was performed using Adam (Kingma and Ba, 2014) with learning rate 0.001, $\beta_1 = 0.9$, $\beta_2 = 0.999$ and $\epsilon = 1 \times 10^{-8}$.

3.4 CONCLUSION

This chapter is an investigation into the consequences of making an assumption about how the test set will differ from the training set. Once we have expressed this assumption, we are able to build a probabilistic model and define a loss function. In this work, we have focused on the assumption that we have a budget and that its allocation affects the noise our model sees at test time.

When we recognise that an application may involve a budget it is better for it to be directly included in the model, allowing it to be accommodated. We have presented a generic method to build a budget into the forward model that is amenable to gradient optimisation, allowing it to scale to high-dimensional problems, online problems or problems with very large datasets.

On a toy digit rotation problem we were able to demonstrate how this method composes with a statistic network to infer rotations and prescribe a budget over which digits to observe at test time. Our experiments showed that allocating a budget according to the gradient-optimised split could be markedly cheaper than allowing an uninformed uniform allocation of resource. As machine learning systems grow and become more integrated with real world problems, we expect that incorporating various sources of data and deciding which data to include at test time will become more common. For example, a predictive service may want to incorporate many different statistics or features from many sources in order to maximise some reward. Our method would be able to identify the most informative features while also providing a well-tuned model that uses those features.

However, learning the rotation of digits was a toy problem. There is no demand for a service that can infer a common rotation between a set of noisy digits. In an attempt to discuss a real world application, we turned to a real dataset of multi-modal information from the Tropical Atmospheric Ocean (TAO) sensor array. A common and relevant problem in sensor arrays is the imputation of missing observations as sensors can never be perfectly reliable in a hostile real world environment. Imputation is often crucial when performing simple analyses which routinely fail in the presence of missing data. Our method provides a basis for the placement or calibration of buoy sensors such that we can most cheaply impute missing data. We found that on average this could save 13% of total budget cost compared to uniform allo-

cation over sensors, or up to 35% on one imputation problem. Unfortunately, many of the imputation problems under consideration were difficult to learn. The level of noise and missing data made it difficult to compete with an imputation strategy of predicting the mean for a given feature.

An alternative problem with real world application is deciding the stored bit depth of elements of an image. Both images and video are ubiquitously stored by lossy compression, where patches of pixels are transformed and quantised according to 8 by 8 quantisation matrices. The JPEG standard (Wallace, 1991) defines a set of these quantisation matrices depending on an arbitrary quality value. With our method we were able to define a bit depth budget and learn a quantisation matrix on each image we wished to compress. Using a learnt model of perceptual quality (Prashnani et al., 2018), we ensured that the learnt image quality was competitive with that of the JPEG alternative. In addition, these quantisation matrices were learnt automatically from *single images*, which may be useful in alternative imaging domains, such as multispectral or medical imaging.

There is growing interest in composable probabilistic modelling where the learning can be achieved using the same stochastic gradient methods popular in deep learning. These are commonly considered attractive due to scalability and readily available frameworks. For example, one may consider variational autoencoders (Kingma and Welling, 2014), black-box variational inference (Ranganath et al., 2013), and the graphical framework of stochastic computation graphs (Schulman et al., 2015) that unites such models. This work provides a new element for composing stochastic computation graphs that include a budgeted element and highlights where this could be useful. Applications include areas where we may form a probabilistic model that includes assumptions of the effect of allocated resources.

Resource allocation to feature gathering was previously studied by Richman and Mannor (2016), in which the authors provide methods to produce resource allocations to contexts in the case of linear models. This work provides a more general framework for the cases of nonlinear models, alternative noise parameterisations and massive datasets. This is achieved using flexible parametric models and gradient descent. Results have been demonstrated on a toy problem of MNIST digit rotation, sensor network resource allocation and learning quantisation matrices from single images.

Modern probabilistic modeling is increasingly required to operate on large datasets, and incorporate flexible parametric models (Schulman et al., 2015). In this work, we have demonstrated that we can build such models to incorporate test-time data gathering constraints. By incorporating budget concerns we have demonstrated potential cost savings and proven these in experiment. The formulation of this protocol we

have given here is a useful addition to conditional modelling using stochastic gradient methods for inference.

4

EFFICIENT ARCHITECTURES[1]

4.1 INTRODUCTION

The problem of designing an efficient deep learning system is often approached as architecture design but these individual architectures are devalued when new state of the art designs appear. The variety of networks that can be created is vast. In this chapter we detail a method that *modifies existing networks* to make them more efficient. As a contribution to deep learning, it can be applied regardless of developments in new network design.

As noted in Chapter 2, deep learning research has focused on performance rather than efficiency. The focus was on demonstrating that the system *could* work, rather than solving the problem using a minimum of resources. However, following this, many papers have capitalised on the ways it is possible to make networks more efficient, or designed purpose-built efficient networks. At test time, on small devices the major concern is storing the massive number of parameters that deep networks have accumulated and this is something our method on which our method focuses. Although, while removing parameters, we also find the number of multiply-add (Mult-Add) operations to drop.

It is possible to take a large pre-trained *teacher* network, and use its outputs to aid in the training of a smaller *student* network (Ba and Caruana, 2014) through some *distillation* process. Distillation refers to use of the activations of a *teacher* network to augment the training loss function of a *student* network. By doing this, the student network is more powerful than if it was trained solely on the training data and is closer in performance to the larger teacher network. The lower-parameter student network typically has an architecture that is more shallow, or thinner — by which we mean its filters have fewer channels (Romero et al., 2014) — than the teacher. While it is not possible to arbitrarily approximate any network with another (Urban et al., 2017), the limit in neural network performance is at least in part due to the training algorithm, rather than its representational power.

We take an alternative approach in designing our student networks. Instead of making networks thinner, or more shallow, we take the standard convolutional block

[1]The work in this chapter was published in at NeurIPS 2018 (Crowley et al., 2017) as joint work with Elliot Crowley and full details on the attribution of work presented here are given in Section 4.1.1.

such networks possess and replace it with a *cheaper* convolutional block, keeping the original architecture. For example, in a Residual Network (ResNet) (He et al., 2016a) this standard block is a pair of sequential 3×3 convolutions. We show that for a comparable number of parameters, student networks that retain the architecture of their teacher but have cheaper convolutional blocks outperform student networks with the original blocks and smaller architectures.

As a model compression strategy, this is very effective. At the same time this transformation is easy to implement in any deep learning framework; replacing convolutional blocks is a simple substitution into any existing architecture. Furthermore, the optimisation scheme used on the teacher network can be repeated on the student, making another round of hyperparameter optimisation unnecessary.

In this chapter we demonstrate:

- A simple, stable method for model compression that is applicable to any convolutional architecture and requires minimal extra engineering time while producing networks at state of the art efficiency.
- Greater compression, in either mult-add or parameter cost, than any prior work using model distillation.
- Comprehensive tests at different compression levels to investigate the relationship between model size and top-1 error.
- Experiments on large-scale image classification and segmentation problems to show that the methods presented here are applicable in settings where distillation may not have been tested before.

The cheap convolutional blocks we suggest are described in Section 4.3 as well as an overview of the methods we employ for distillation. In Section 4.4, we train a number of student networks for the task of image classification on the CIFAR-10 and CIFAR-100 (Krizhevsky, 2009) datasets and demonstrate that those with cheap convolutions perform better than shallower student networks for a given parameter cost.

Although it is possible to train the resulting architectures directly, in Section 4.4.2 we demonstrate this is less effective than distilling them from the larger teacher model. Finally, in Section 4.5 we show that our method generalises well for image classification on ImageNet (Deng et al., 2009) and semantic segmentation on the Cityscapes dataset (Cordts et al., 2016).

4.1.1 *Attribution*

This work was completed jointly with Elliot Crowley. The original concept was developed in collaboration. I developed the initial implementation. Elliot developed this implementation further and we discussed how the substitute blocks presented in Section 4.3.2 should be defined. Together we agreed on the blocks presented here. I developed the final implementation of the code implementing the experiments presented in Sections 4.4 and 4.5, which is publicly available[1].

The description of the methods was written in collaboration for the original paper, while I produced Figure 4.3. Supplemental description beyond that in the original paper (Crowley et al., 2017) is my own work.

Of the experimental results presented here:

- The experimental results described in Figure 4.4 and Table 4.3 were produced in experiments run by Elliot Crowley using the code we had jointly developed.
- I produced Figure 4.7 using results from two experiments run by Elliot Crowley using the code we developed. These are also presented in Table 4.4.
- The results detailed in Figures 4.5 and 4.6 are my own work and are not present in the original paper.

The description of experiments given in Section 4.4 is based on that of the original paper (Crowley et al., 2017) and was written by Elliot Crowley, but I have expanded the text to provide additional details. All other text is my own work.

4.2 RELATED WORK

It is likely that the work presented in this chapter would not be possible were it not for the over-parameterisation of neural networks (Denil et al., 2013). This was discussed in detail in Chapter 2.

This work also draws directly from the prior work on network distillation, which was described in Section 2.5.6. We directly apply attention transfer (Zagoruyko and Komodakis, 2017) for distillation. It was found to work better than knowledge distillation (Ba and Caruana, 2014; Hinton et al., 2016) by experiment, as shown in Section 4.4.

To modify existing network architectures, we focus on substituting repeating structures, which will be explained in more detail in Section 4.3. Our substitutes are based

[1] The code implementing all experiments presented here can be found at `https://github.com/BayesWatch/pytorch-moonshine`

on motifs common in modern neural networks, such as separable convolutions and bottlenecks (Ioffe and Szegedy, 2015; Chollet, 2016; Xie et al., 2017; Howard et al., 2017). These were discussed in more detail in Section 2.5.

4.3 COMPRESSION WITH CHEAP CONVOLUTIONS

Given a large, deep network that performs well on a given task, we are interested in compressing that network so that it uses fewer parameters. A flexible and widely applicable way to reduce the number of parameters in a model is to replace all its convolutional layers with a cheaper alternative. Doing this replacement invariably impairs performance when this reduced network is trained directly on the data. Fortunately, we are able to demonstrate that modern distillation methods enable the cheaper model to have performance closer to the original large network.

4.3.1 *Distillation*

For this paper, we utilise and compare two different distillation methods for learning a smaller student network from a large, pre-trained teacher network: knowledge distillation (Ba and Caruana, 2014; Hinton et al., 2016) and attention transfer (Zagoruyko and Komodakis, 2017).

Knowledge Distillation Let us denote the cross entropy of two probability vectors $\mathbf{p}$ and $\mathbf{q}$ as $\mathcal{L}_{CE}(\mathbf{p}, \mathbf{q}) = -\sum_k p_k \log q_k$. Assume we have a dataset of elements, with one such element denoted $\mathbf{x}$, where each element has a corresponding one-hot class label: $\mathbf{y}$. Given $\mathbf{x}$, we have a trained teacher network $\mathbf{t} = \text{teacher}(\mathbf{x})$ that outputs the corresponding logits, $\mathbf{t}$; likewise we have a student network that outputs logits $\mathbf{s} = \text{student}(\mathbf{x})$. To perform knowledge distillation we train the student network to minimise the following loss function (averaged across all data items):

$$\mathcal{L}_{KD} = (1-\alpha)\mathcal{L}_{CE}(\mathbf{y}, \sigma(\mathbf{s})) + 2\alpha T^2 \mathcal{L}_{CE}\left(\sigma\left(\frac{\mathbf{t}}{T}\right), \sigma\left(\frac{\mathbf{s}}{T}\right)\right), \tag{4.1}$$

where $\sigma(.)$ is the softmax function, T is a temperature parameter and α is a parameter controlling the ratio of the two terms in the sum. The first term is a standard cross entropy loss penalising the student network for incorrect classifications. The second term is minimised if the student network produces outputs similar to that of the teacher network. The idea being that the outputs of the teacher network contain additional beneficial information beyond just a class prediction.

Some theoretical justification for this intuition can be provided from information theory. Each class label comes from a multinomial distribution and as such can provide a maximum of $-\log_2(\frac{1}{N})$ bits of information for N different labels (Cover and Thomas, 2006, p.14). In contrast, the distribution of the logits, or attention maps, used in distillation are continuous and therefore can potentially have unbounded entropy. For example, if it were Gaussian distributed, we can define entropy as $H(X) = \frac{1}{2}(2\pi e\sigma^2)$ and we can see that it becomes unbounded as σ increases: $\lim_{\sigma\to\inf} \frac{1}{2}(2\pi e\sigma^2) = \inf$ (Cover and Thomas, 2006, p.263). Although, while this reasoning relies on differential entropy, it can be assumed that it is also true for a limiting density of points, and is indeed hypothesised as the reason knowledge distillation works by Hinton et al. (2016).

Attention Transfer Consider some choice of layers with indices $i \in \{1, 2, ..., N_L\}$ in a teacher network and the corresponding layer in the student network. At each chosen layer i of the teacher network, collect the spatial map of the activations for channel j into the vector $\mathbf{a}_{ij}^t$. Let A_i^t collect $\mathbf{a}_{ij}^t$ for all channels j. Likewise, for the student network we correspondingly collect into A_i^s and $\mathbf{a}_{ij}^s$.

Now, given some choice of mapping $\mathbf{f}(A_i)$ that maps each collection of the form A_i into a vector, attention transfer involves learning the student network by minimising the following expression:

$$\mathcal{L}_{AT} = \mathcal{L}_{CE}(\mathbf{y}, \sigma(\mathbf{s})) + \beta \sum_{i=1}^{N_L} \left\| \frac{\mathbf{f}(A_i^t)}{\|\mathbf{f}(A_i^t)\|_2} - \frac{\mathbf{f}(A_i^s)}{\|\mathbf{f}(A_i^s)\|_2} \right\|_2, \tag{4.2}$$

where β is a hyperparameter. Zagoruyko and Komodakis (2017) recommended using $\mathbf{f}(A_i^{\{s,t\}}) = (1/C_i) \sum_{j=1}^{C_i} (\mathbf{a}_{ij}^{\{s,t\}})^2$, where C_i is the number of channels at layer i. In other words, the loss targets the difference in the spatial map of average squared activations, where each spatial map is normalised by the overall activation norm.

Let us examine the loss in Equation 4.2 further. The first term is again a standard cross entropy loss. The second term, however, ensures the spatial distribution of the student and teacher activations are similar at selected layers in the network, the explanation being that both networks are then *"paying attention"* to the same things at those layers.

This "attention" is so named because the response of elements in the attention map A_i^t can be mapped back to the input image. When doing so, larger activations give some indication as to the features used by the neural network in classification. An illustration of this is provided in Figure 4.1, in which the attention values $\mathbf{f}(A_i)$ are plotted over an input image as a heatmap, respecting their relative x, y positions projected onto the input space.

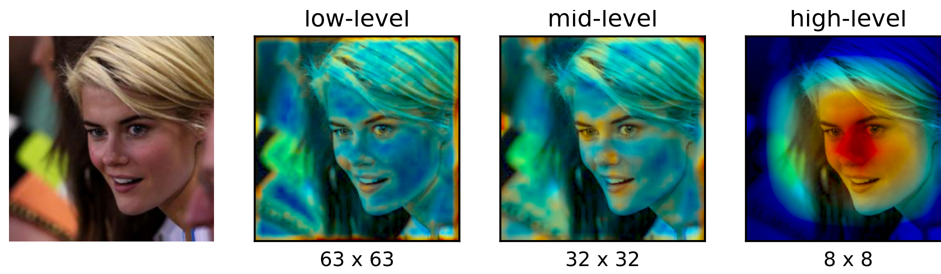


Figure 4.1: "Sum of absolute values attention maps F_{sum} over different levels of a network trained for face recognition. Mid-level attention maps have higher activation level around eyes, nose and lips, high-level activations correspond to the whole face." (Zagoruyko and Komodakis, 2017)

However, this is a crude method to gauge the response of the network. A more principled way to express the sensitivity of the prediction to the input would be to use the gradient of the loss with respect to each input pixel, $J = \frac{\delta}{\delta x}\mathcal{L}(\mathbf{W}, x)$. We could devise a loss where the student and teacher, each having weights $\mathbf{W_S}$ and $\mathbf{W_T}$, attempt to match this gradient over the whole input space. The resulting second order partial derivative can be evaluated using the same automatic gradient methods widely used in deep learning. This is similar to a double backpropagation method suggested by Drucker and LeCun (1992).

Unfortunately, despite the elegance of this solution, using the spatial attention maps at arbitrary places in the network was found to work better in experiments by Zagoruyko and Komodakis (2017). Experimenting with a Network-In-Network (Lin et al., 2013) on CIFAR-10, they found it to only perform as well as knowledge distillation (Hinton et al., 2016; Ba and Caruana, 2014).

4.3.2 *Cheap Convolutions*

The expense incurred in making a prediction is computational. We focus here on two computational costs: the cost of storage, counting the number of floating point values used to store the parameters, and the cost of processing, by counting the mult-adds executed in making a prediction. What we aim to produce are cheap convolutional architectures; those where a forward pass can both be stored in fewer bytes and consume fewer mult-adds when processing.

Convolution is one of the major components that makes large-scale deep learning on natural images possible. It involves passing a filter kernel $K \in \mathbb{R}^{m \times n}$ over all locations in an input array, as illustrated in Figure 4.2. Convolutions are well-suited to processing image data due to the *equivariance* of the filter over the entire input space (Goodfellow et al., 2016, p.326) because we do not know which part of the image we may need to attend to. Reusing the same filter many times is also a form of pa-

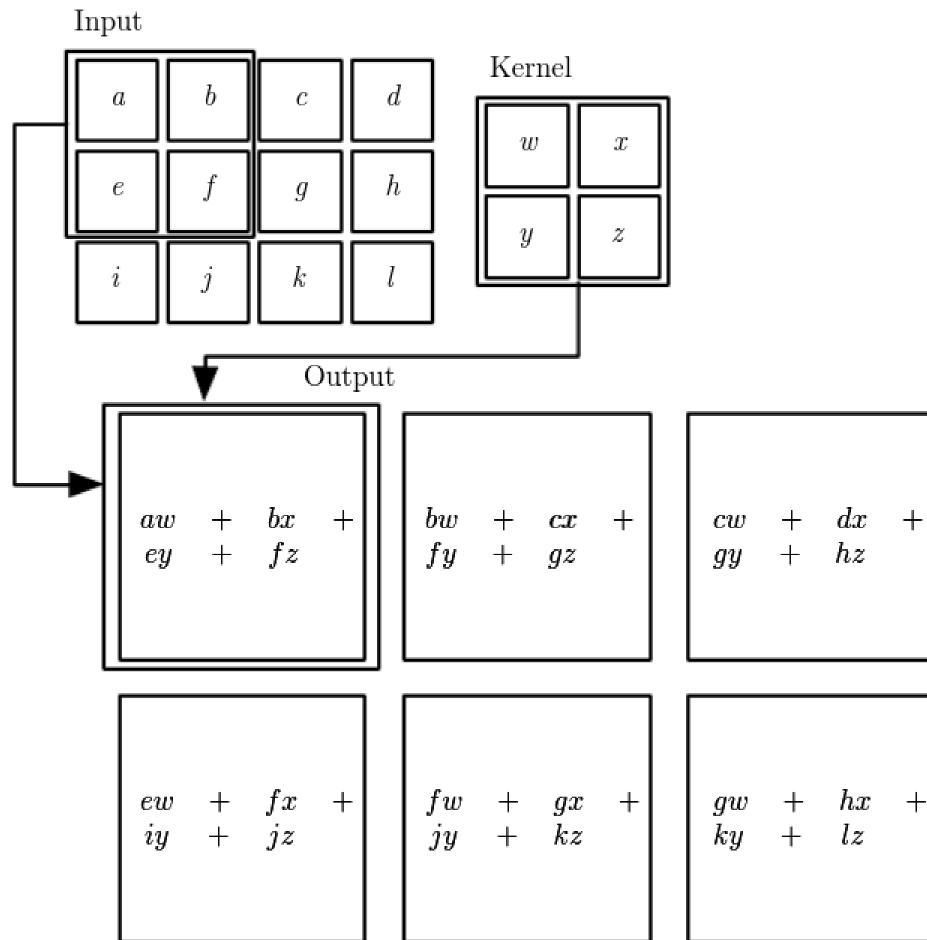


Figure 4.2: Example of a convolution on a 2D input array, passing a 2×2 filter over the input space and returning 6 output values (Goodfellow et al., 2016, p.325).

rameter sharing that enables us to represent a transformation on a high-dimensional tensor efficiently.

However, AlexNet (Krizhevsky et al., 2012a), the first architecture to outperform more traditional feature driven computer vision methods on the ImageNet (Deng et al., 2009) object recognition challenge, the deep network that was first able to beat other computer vision methods, was composed with two very large fully connected layers in the final parts of the network. As fully connected layers consume N^2 parameters, where N is the number of activations, these layers held most of the parameters in the network.

Recent networks have shown that this is not necessary. For example, the all-convolutional network was able to achieve greater performance using no fully-connected layers at all (Springenberg et al., 2014). The trend has moved towards many repeating blocks composed of convolutional layers, often with skip

connections, where the vast majority of the parameters are found in the convolutional layers (He et al., 2016a).

This abstracts network design into a problem focused on the design of arbitrary blocks, such as those containing the same spatial size, as in Table 4.2, or a sequence of two convolutions, as in Table 4.1. Here, we present several convolutional blocks that may be introduced in place of a block of two sequential convolutions in a network to substantially reduce its parameter cost[2].

First, let us consider a standard two dimensional convolutional layer that contains N_{out} filters, each of size $N_{in} \times k \times k$. N_{out} is the number of channels of the output layer, N_{in} is the number of channels of the input layer, and $k \times k$ is the kernel size of each convolution. In modern networks it is almost always the case that $N_{in} \leqslant N_{out}$; in other words, that the number of channels increases as we pass through convolutional layers (He et al., 2016a).

Let $N = \max(N_{in}, N_{out})$. The parameter cost of this layer is $N_{in}N_{out}k^2$ and is bounded by N^2k^2. In a typical residual network, a block contains two such convolutions. We will refer to this as a *Standard* block S and it is outlined in Table 4.1.

An alternative approach is to separate each convolution into g groups, as shown in Figure 4.3a. If we express the convolution operation, c, on a 4D tensor $\boldsymbol{\mathcal{A}}_l \in \mathbb{R}^{1 \times N_{in} \times H \times W}$, height H and width W, using weight tensor $\boldsymbol{\mathcal{W}}_l \in \mathbb{R}^{N_{out} \times N_{in} \times k_H \times k_W}$, kernel height k_H and width k_W, at layer l as

$$\boldsymbol{\mathcal{A}}_{l+1} = c(\boldsymbol{\mathcal{A}}_l, \boldsymbol{\mathcal{W}}_l) \tag{4.3}$$

then a grouped convolution, with $g = N_{in}$, is equivalent to the weight parameterisation $\boldsymbol{\mathcal{V}}_l$ with elements $v_l^{ijlk} = w_l^{ijlk}\mathbf{1}_{i=j}$, where w_l^{ijlk} are the elements of $\boldsymbol{\mathcal{W}}_l$. Similarly, a pointwise convolution uses a weight tensor $\boldsymbol{\mathcal{P}}_l$ with the constraint that $\boldsymbol{\mathcal{P}}_l \in \mathbb{R}^{N_{in} \times N_{out} \times 1 \times 1}$. Using these definitions, we can express a separable convolution as:

$$\boldsymbol{\mathcal{A}}_{l+1} = c(c(\boldsymbol{\mathcal{A}}_l, \boldsymbol{\mathcal{V}}_l), \boldsymbol{\mathcal{P}}_l) \tag{4.4}$$

By restricting the convolutions to only mix channels within each group, we obtain a substantial reduction in the number of parameters for a grouped computation: for example, for $N_{in} = N_{out} = N$ the cost changes from N^2k^2 for a standard layer to g groups of $({}^{N^2}/g)\,k^2$ parameter convolutions, hence reducing the parameter cost by a factor of g. Using a pointwise convolution in sequence provides cross-group mixing, with a N^2 parameter cost (when $N_{in} \neq N_{out}$ the change in channel size occurs across

[2]The parameters introduced by batch normalisation are negligible compared to those in the convolutions. However, they are included for completeness in Table 4.1.

this pointwise convolution). We refer to this substitution operator as $G(g)$ (grouped convolution with g groups, followed by a pointwise convolution) and illustrate it in Figure 4.3b.

He et al. (2016a) introduce a bottleneck block, along with the ResNet architecture, which we have parameterised and denoted as $B(b)$: the input first has its channels decreased by a factor of b via a pointwise convolution before a full convolution is carried out. Finally, another pointwise convolution brings the representation back up to the desired N_{out}. We can reduce the parameter cost of this block even further by replacing the full convolution with a grouped one; the *Bottleneck Grouped + Pointwise* block is referred to as $BG(b, g)$ and is illustrated in Figure 4.3b.

These substitute blocks are compared in Table 4.1 where their computational costs are given. In practice, by varying the bottleneck size and the number of groups, network parameter numbers may vary over two orders of magnitude; enumerated examples illustrating this are given in Table 4.3.

Using grouped convolutions and bottlenecks are common methods for parameter reduction when designing a network architecture. Both are easy to implement in any deep learning framework. Sparsity inducing methods (Han et al., 2015), or approximate layers (Yang et al., 2015), may also provide advantages, but these are complementary to the approaches here. More structured reductions such as grouped convolutions and bottlenecks can be advantageous over sparsity methods in that the efficient structure is easily represented and leveraged. In contrast, the efficiency benefits of a sparse structure depend on what sparse structure happens to be discovered. In the following sections, we demonstrate that using these proposed blocks with effective model distillation allows for substantial compression with minimal reduction in performance.

4.4 CIFAR EXPERIMENTS

In this section we train and evaluate a number of student networks, each distilled from the *same* large teacher network. Experiments are conducted for both the CIFAR-10 and CIFAR-100 datasets. We apply knowledge distillation and attention transfer. We also train the networks without any form of distillation (i.e. from scratch) to observe whether the distillation process is necessary to obtain good performance. In this way we demonstrate that the high performance comes from the distillation, and cannot be achieved by directly training the student networks using the data.

For comparison we also study student networks with smaller architectures (i.e. fewer layers/filters) than the teacher. This enables us to test if the block transformations we propose are key, or it is simply a matter of distilling networks with smaller

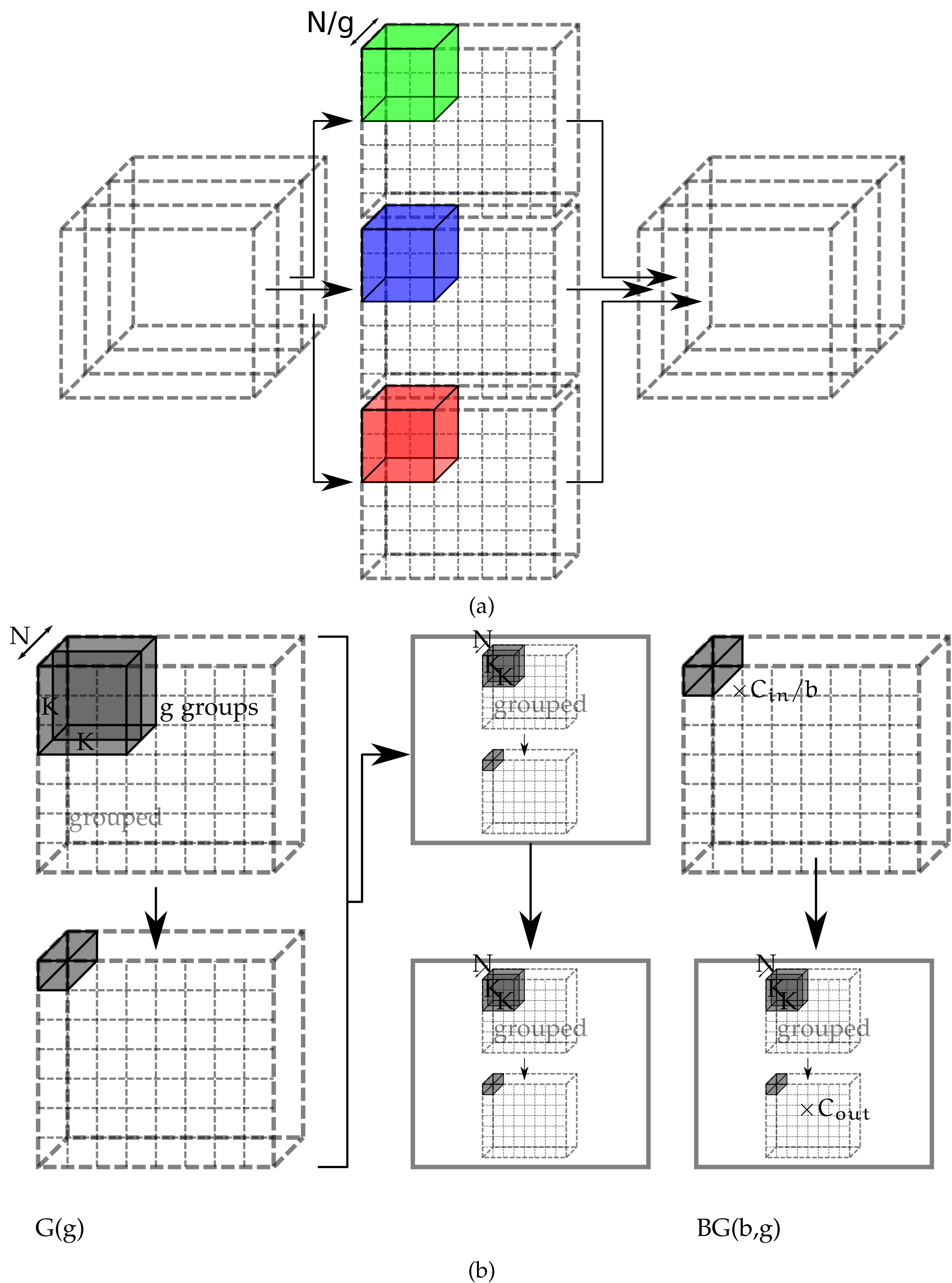


Figure 4.3: In (a), a grouped convolution operates by passing independent filters over the tensor after it is separated into g groups over the channel dimension; as each of the g filters needs only to operate over N/g channels, this reduces the parameter cost of the layer by a factor of g. These can be composed into the blocks illustrated in (b). The *Grouped + Pointwise* block ($G(g)$) substitutes a $k \times k$ convolution with a grouped convolution followed by a pointwise (1×1) convolution, repeating this twice. To reduce parameters further, a pointwise *Bottleneck* can be used before the Grouped + Pointwise convolution ($BG(b, g)$).

Table 4.1: Convolutional Blocks used in this paper: a standard block S, a grouped + pointwise block G, a bottleneck block B, and a bottleneck grouped + pointwise block BG. Conv refers to a $k \times k$ convolution. GConv is a grouped $k \times k$ convolution and Conv1x1 is a pointwise convolution. Blocks use pre-activations (He et al., 2016b): all convolutions are preceded by a batch-norm layer + a ReLU activation. We assume that the input and output to each block has N channels and that channel size does not change over a particular convolution unless written out explicitly as $(x \rightarrow y)$. Where applicable, g is the number of groups in a grouped convolution and b is the bottleneck contraction. We give the cost of the convolutions in each block in terms of these parameters. The batch-norm (denoted BN) cost at test time is also given, but is markedly smaller.

Block	S	G(g)	B(b)	BG(b, g)
Structure	Conv Conv	GConv (g) Conv1x1 GConv (g) Conv1x1	Conv1x1($N \rightarrow \frac{N}{b}$) Conv Conv1x1($\frac{N}{b} \rightarrow N$)	Conv1x1($N \rightarrow \frac{N}{b}$) GConv(g) Conv1x1($\frac{N}{b} \rightarrow N$)
Conv Params	$2N^2k^2$	$2N^2(\frac{k^2}{g}+1)$	$N^2(\frac{k^2}{b^2}+\frac{2}{b})$	$N^2(\frac{k^2}{gb^2}+\frac{2}{b})$
BN Params	4N	8N	$N(2+\frac{4}{b})$	$N(2+\frac{4}{b})$

numbers of parameters. We compare the smaller student architectures with student architectures implementing cheap, substitute convolutional blocks, but with the same architecture as the teacher. The different convolutional blocks are summarised in Table 4.1 and the student networks are described in detail in Section 4.4.1. Results are given in Table 4.3 and Figure 4.4. These results are discussed in detail in Section 4.4.2.

4.4.1 *Network Descriptions*

For our experiments we utilise the Wide Residual Network (WRN) architecture (Zagoruyko and Komodakis, 2016). The bulk of the network lies in its {*conv*2, *conv*3, *conv*4} groups and the network depth d determines the number of convolutional blocks n in these groups as $n = (d-4)/6$. This structure is described in Table 4.2. The network width, denoted by k, affects the channel size of the filters in these blocks. Note that when we employ attention transfer, the student and teacher outputs of groups {*conv*2, *conv*3, *conv*4} are used as $\{A_1, A_2, A_3\}$ in the second term of Equation (4.2) with $N_L = 3$.

For our teacher network we use WRN-40-2 (a WRN with depth 40 and width multiplier 2 with standard (S) blocks. 3×3 kernels are used for all non-pointwise convolutions in our student and teacher networks unless stated otherwise.

For our student networks we use:

group	output size	structure
$\mathtt{conv1}$	$16 \times 32 \times 32$	$1\times$ Conv3x3(N = 16)
$\mathtt{conv2}$	$16k \times 32 \times 32$	$n\times$ Block(N = 16k)
$\mathtt{conv3}$	$32k \times 16 \times 16$	$n\times$ Block(N = 32k)
$\mathtt{conv4}$	$64k \times 8 \times 8$	$n\times$ Block(N = 64k)
$\mathtt{pool}$	$64k \times 1 \times 1$	8×8 avg-pool
$\mathtt{fc}$	$\mathtt{classes}$	$64k \times \mathtt{classes}$ fully connected

Table 4.2: Summary of the Wide ResNet structures used in experiments; matching those in Zagoruyko and Komodakis (2017). The bulk of the parameters are in {conv2, conv3, conv4} which each consist of n blocks with channel width N controlled by k. We explore the effect of substituting these blocks with cheaper alternatives. classes refers to the number of object classes which is, perhaps unsurprisingly, 10 for CIFAR-10 and 100 for CIFAR-100.

- WRN-40-1, 16-2, and 16-1 with S blocks. These are student networks that are thinner and/or more shallow than the teacher and represent typical student networks used.
- WRN-40-2 with S blocks where the 3×3 kernels have been replaced with 2×2 dilated kernels (as described in Yu and Koltun (2016)). This allows us to see if it possible to naively reduce parameters by effectively zeroing out elements of standard kernel.
- WRN-40-2 using a bottleneck block B with $2\times$ and $4\times$ channel contraction (b), as shown in Table 4.1.
- WRN-40-2 using a grouped + pointwise block G for group sizes $g \in \{2, 4, 8, 16, N/16, N/8, N/4, N/2, N\}$, where N is the number of channels in a given block. This allows us to explore the spectrum between full convolutions ($g = 1$) and fully separable convolutions ($g = N$).
- WRN-40-2 with a bottleneck grouped + pointwise block BG. We use $b = 2$ with groups sizes of $g \in \{2, 4, 8, 16, M/16, M/8, M/4, M/2, M\}$, where $M = N/b$ is the number of channels *after the bottleneck*. We use this notation so that $g = M$ represents fully separable convolutions and we can easily denote divisions thereof. $BG(4, M)$ is also used to observe the effect of extreme compression.

Blocks BG and G are illustrated in Figure 4.3b.

Implementation Details To demonstrate that we can reuse the hyperparameters used for training the teacher model, we use the training protocol described by Zagoruyko

and Komodakis (2017). For training we used minibatches of size 128. Before each minibatch, the images were padded by 4×4 zeros, and then a random 32×32 crop was taken. Each image was left-right flipped with a probability of a half. Networks were trained for 200 epochs using SGD with momentum fixed at 0.9 with an initial learning rate of 0.1. The learning rate was reduced by a factor of 0.2 at the start of epochs 60, 120, and 160. For knowledge distillation we set α to 0.9 and used a temperature of 4. For attention transfer β was set to 1000. The code to reproduce these experiments is publicly available[3].

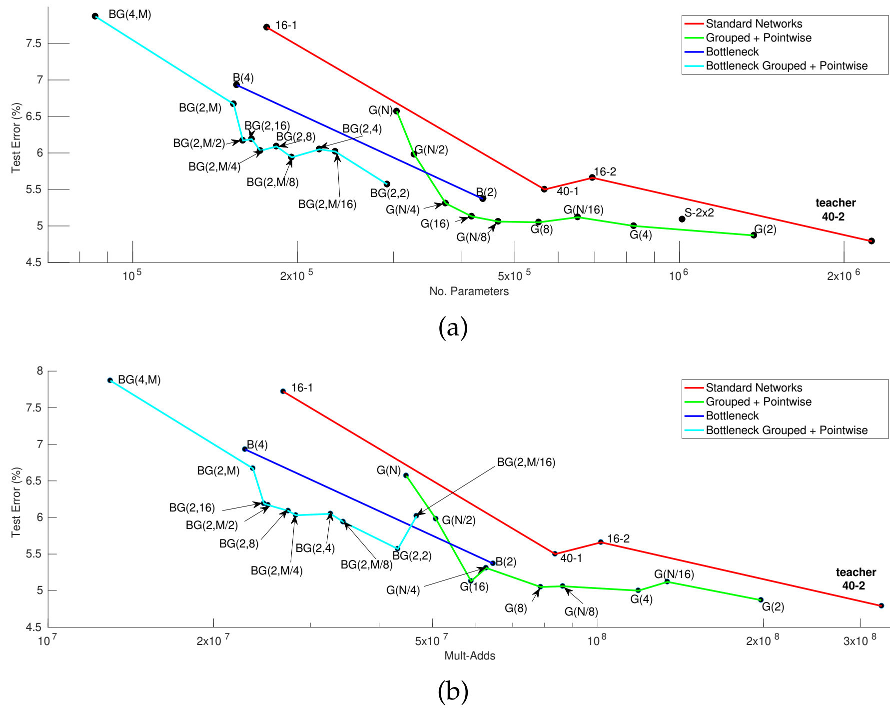
4.4.2 *Analysis and Observations*

Figure 4.4a compares the parameter cost of each student network (on a log scale) against the test error on CIFAR-10 obtained with attention transfer. On this plot, the ideal network would lie in the bottom-left corner (few parameters, low error). What is fascinating is that almost every network with the same architecture as the teacher, but with cheap convolutional blocks (those on the blue, green, and cyan lines) performs better for a given parameter budget than the reduced architecture networks with standard blocks (the red line). $BG(2,2)$ outperforms 16-2 (5.57% vs. 5.66%) despite having considerably fewer parameters (287K vs. 692K). Several of the networks with BG blocks both significantly outperform the smaller WRN-16-1 network, using the original convolutional blocks, while using fewer parameters.

It is encouraging that significant compression is possible with only small losses: several networks perform almost as well as the teacher with considerably fewer parameters – $G(N/8)$ has an error of 5.06%, close to that of the teacher (4.79%), but has just over a fifth of the parameters (0.45M versus 2.24M). $BG(2, M/8)$ has less than a tenth of the parameters of the teacher (0.19M versus 2.24M), for a cost of 1.15% increase in error. Even simply switching all convolutions with smaller, dilated equivalents ($S - 2 \times 2$) allows one to use half the parameters for a similar performance (1.01M versus 2.24M, with error 5.09%).

An important lesson can be learnt regarding grouped + pointwise convolutions. They are often used in their fully separable (Chollet, 2016) form ($g = N$). However, the networks with half, or quarter that number of groups perform substantially better for a modest increase in parameters. $G(N/4)$ has 363K parameters compared to the 294K of $G(N)$ but has an error that is 1.26% lower, which is a substantial fraction of the top-1 error on this problem. The number of groups is an easy parameter to tune to trade some performance for a smaller network. Grouped + pointwise convolutions also work well in conjunction with a bottleneck of size 2, although for bottlenecks

[3] https://github.com/BayesWatch/pytorch-moonshine



(a)

(b)

Figure 4.4: Test Error vs. (a) Number of parameters and (b) mult-adds for student networks learnt with attention transfer on CIFAR-10. Note that the x-axes are log-scaled. Points on the red curve correspond to networks with S convolutional blocks and reduced architectures. All other networks have the same WRN-40-2 architecture as the teacher but with cheap convolutional blocks: G (green), B (blue), and BG (cyan). The blocks are described in Table 4.1. Notice that the student networks with cheap blocks outperform those with smaller architectures and standard convolutions for a given parameter budget or mult-add budget.

with fewer channels the error increases significantly, as can be seen for $BG(4, M)$. Despite this, it is still of comparable performance to 16-1 with half the parameters. Similar trends are observed for CIFAR-100 in Table 4.3, lending weight to results on CIFAR-10.

We also observe that training a student with attention transfer (AT) is substantially better than using knowledge distillation, or simply training from scratch. Consider Table 4.3, which shows the attention transfer errors of Figure 4.4 (the AT column) alongside those of networks trained with knowledge distillation (KD), and no distillation i.e. from scratch (Scr) for CIFAR-10 and CIFAR-100. In all cases, the student network trained with attention transfer is better than the student network trained by itself, ie from scratch (Scr) – the distillation process appears to be necessary. In line with the results of Zagoruyko and Komodakis (2017), we also find that KD yields higher error than AT. Some performances are particularly impressive; on CIFAR-10, for $G(2)$ blocks the error is only 0.08% higher than the teacher despite the network having 60% of the parameters.

These results support our claim that greater model compression through distillation is possible by substituting the convolutional blocks in a network, rather than by shrinking its architecture. We have also demonstrated that the blocks outlined in Table 4.1 are suitable substitutes. By observing Figure 4.4b we can also see that our networks with cheap, substitute blocks utilise fewer multiply–accumulate operations (mult-adds) than their standard equivalents, which is often used as an indication of potential runtime speed (Howard et al., 2017). However, it is worth noting that actual runtime on a given platform or device is dependent on specifics (memory paging, choice of libraries etc.), so mult-adds are not always fully indicative of runtime, but are a decent approximation in a platform/implementation-agnostic setting.

4.4.3 *Optimisation Dynamics*

Attention transfer adds additional components to the loss function (shown in Equation 4.2), corresponding to the normalised difference in spatial means at certain layers. We might ask how these components compare to the cross-entropy loss being minimised for classification performance, or how these components progress during training.

In Figure 4.5, these different components are illustrated. The earliest component in the network, at Block 1, is minimised first, with the component at Block 2 following it during training. The final component, at Block 3, is never minimised to the same degree. The cross-entropy loss is also illustrated, and we see that after scaling by the

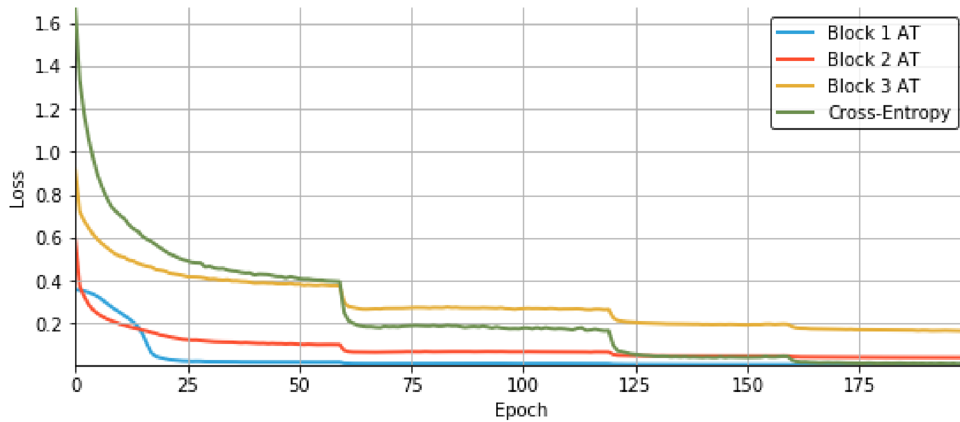


Figure 4.5: The progression of the different components of the loss during training, when using attention transfer. "Blocks" refer to the blockwise structure of a Wide ResNet (Zagoruyko and Komodakis, 2016), higher indices proceeding deeper into the network, the student network used a grouped+bottleneck block with the same architecture of depth 40 and width 2. Each AT component has been scaled by the appropriate β value used during training.

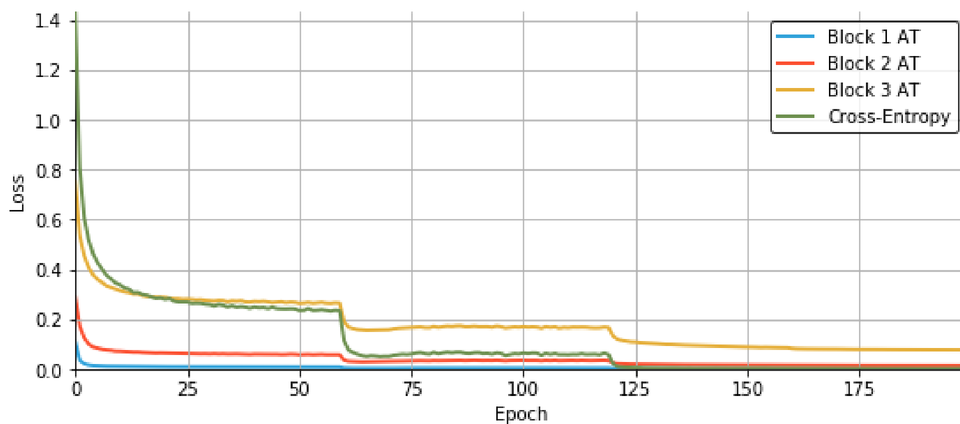


Figure 4.6: Progress of components of the loss function during training, as in Figure 4.5, but the student used here has precisely the same structure as the teacher.

default setting of $\beta = 10^3$, the attention transfer components are approximately the same size; most components begin within the same order of magnitude.

It is unclear whether we observe this behaviour in the AT loss corresponding to block 3 due to a lack of representational capacity in the student network, or only due to accumulated errors between the student and teacher networks. To investigate this, we can observe how the progression changes when the student and teacher networks have the same capacity.

In Figure 4.6 learning curves showing the progression of different components of the loss function are shown as the epochs of training progress. A WRN (Zagoruyko and Komodakis, 2016) was trained with attention transfer, using a teacher and student with the same architecture: depth 40 and width factor 2, with standard convolu-

tional blocks[4]. While the losses approach zero more quickly, they do not all reach zero in this experiment either, which lends some weight to the hypothesis that differences in the activations earlier in the network make it more difficult to match activation maps later.

4.5 SCALING AND GENERALISATION

Section 4.4 demonstrates the effectiveness of *cheapening* convolutions for CIFAR classification. In this section, we apply this method to two further problems. Firstly, in Section 4.5.1 we examine whether the observed benefits hold for large-scale image classification on ImageNet (Deng et al., 2009), where there are far more classes (1000), and the images are significantly larger. Secondly, in Section 4.5.2 we cheapen the convolutions of a network trained for semantic segmentation.

4.5.1 *ImageNet*

Our experiments use a pre-trained ResNet-34 (He et al., 2016a) (21.8M parameters) as a teacher and we train several networks using AT, as we observed it to universally improve performance in the experiments of Section 4.4. We compare student networks that have the architecture of ResNet-34 with *cheaper* convolutions to those that have reduced architectures and full convolutions. Note that the bulk of the parameters in a ResNet are contained in four groups, as opposed to the three groups of a Wide ResNet. The following student networks were chosen for comparison against the work of Zagoruyko and Komodakis (2017):

- (i) ResNet-18,
- (ii) ResNet-18 with the channel widths of the last three groups halved (Res18-0.5 in Table 4.4),
- (iii) ResNet-34 with each convolutional block replaced by a $G(N)$ block,[5]
- (iv) ResNet-34 with each convolutional block replaced by a $G(4)$ block.

Validation errors for these networks are available in Table 4.4.

[4] It is worth noting that this is a partial replication of the work of Furlanello et al. (2018), showing distillation on a model using the same structure as the teacher. We confirmed their observation of a 0.1% increase in accuracy on the student model, over the teacher.

[5] As the convolutional blocks in the teacher do not use pre-activations, the G blocks used here are modified accordingly (BN + ReLU now come after each convolution). This also applies to the networks in Section 4.5.2.

Consider Res34-G(N) and Res18-0.5, which both have roughly the same parameter cost (~3M). After distillation, the former has a significantly lower *top-5 error* (10.66% vs. 15.02%), the percentage of examples where the top-5 predictions contained the correct class. This again supports our claim that is is preferable to cheapen convolutions rather than shrink the network architecture. Res34-G(N) trained from scratch has a noticeably higher top-5 error (12.26%), which shows that it benefits from distillation. Conversely, distillation makes Res18-0.5 slightly worse, suggesting that it has no further representational capacity.

Res34-G(4) similarly outperforms Res18 (these are roughly similar in cost at 8.1M and 11.7M parameters respectively), although in this case the latter does benefit from distillation. It is intriguing that Res34-G(4) trained from scratch is actually on par with the original teacher (having a 0.12% lower top-1 error, and a 0.05% higher top-5 error), despite having 13 million fewer parameters. This generalisation capability of grouped convolutions in networks has been previously observed by Ioannou et al. (2017). Distillation is able to push its performance slightly further to the point that its top-5 error surpasses that of the teacher (8.43% vs. 8.57%).

Implementation Details We again mirror the training protocol used by Zagoruyko and Komodakis (2017) in experiments on ImageNet. Models were trained for 100 epochs using SGD with an initial learning rate of 0.1, momentum of 0.9, and weight decay of 10^{-4}. The learning rate was reduced by a factor of 10 every 30 epochs. Minibatches of size 256 were used across 4 GPUs. When trained with a teacher, an additional AT loss was used with the outputs of the four groups of each ResNet. β was set to 750 so that the total contribution of the AT loss was the same as in Section 4.4.

State of the Art While this is a generic compression method, and we could just as easily apply this to any network, it is worth comparing the networks we have produced to recent efficient networks in the literature. As efficiency is not a single metric, we compare the relative top 1 error, parameter count, and multiply-add operations.

Figure 4.7 illustrates where the networks we have trained are placed on this error-parameter count trade-off. The ResNet-34 that is the basis of the experimental design is not specifically designed for efficiency so it is perhaps surprising to see our networks comparing favourably to MobileNet (Howard et al., 2017), a recent architecture explicitly designed with efficiency in mind.

It can also be noted that the networks learnt by architecture search, such as NASNet (Zoph et al., 2017), lie along a relatively small range of parameter or Mult-Add operations. This is an artefact of the architecture search process: to obtain efficient

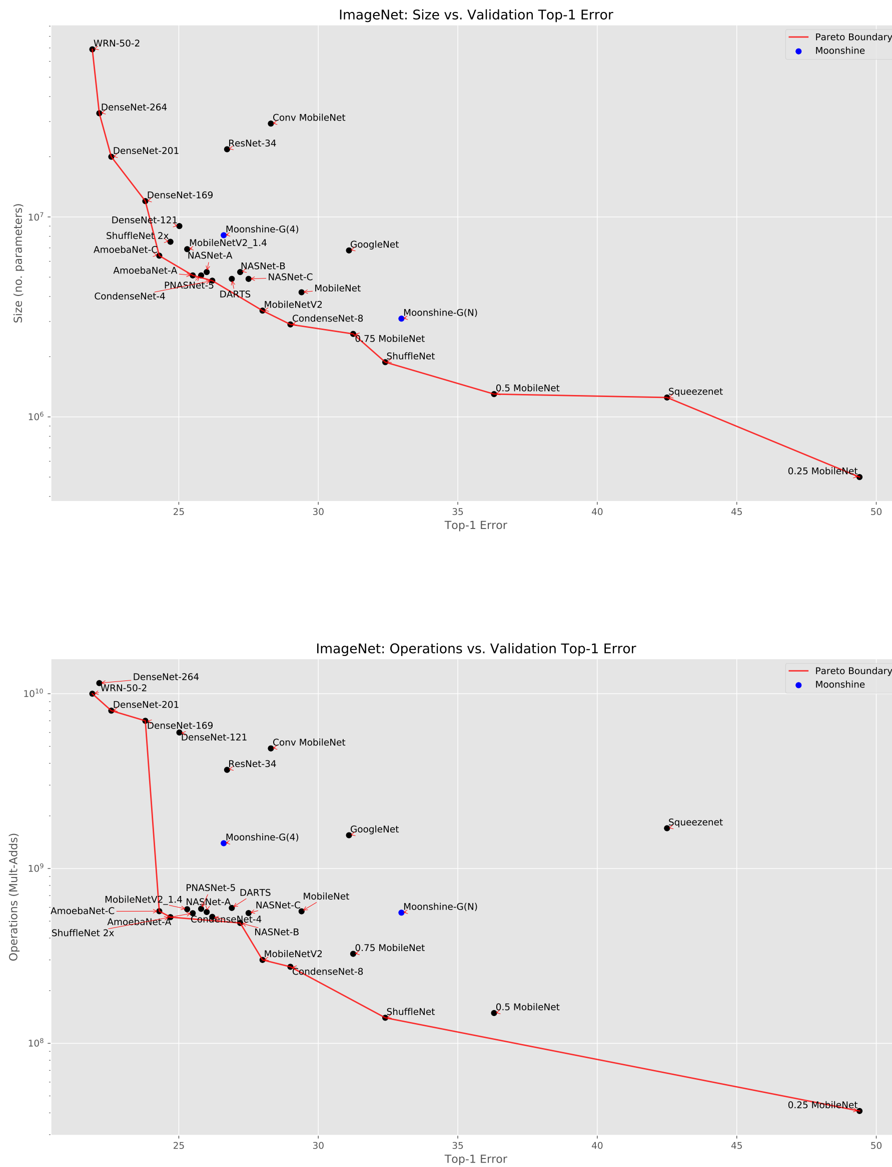


Figure 4.7: Comparing top-1 error (%) to parameter count (top) and multiply-add operations (bottom). The two ResNet-34 networks presented in Table 4.4 are tagged "Moonshine". Networks compared against are recent networks presented in the literature as efficient, including: MobileNet (Howard et al., 2017), GoogleNet (Szegedy et al., 2015), SqueezeNet (Iandola et al., 2016), ShuffleNet (Zhang et al., 2017a), NASNet (Zoph et al., 2017), PNASNet (Liu et al., 2017), AmoebaNet (Real et al., 2017) and DARTS (Liu et al., 2018).

networks, the black box optimiser used would typically be set a budget in parameter count or mult-add operations. We might also note that networks such as ShuffleNet (Zhang et al., 2017a) or MobileNet (Howard et al., 2017) could likely be compressed using our method to extend the Pareto boundary further towards the bottom left of the graph.

4.5.2 *Semantic Segmentation*

We have shown that cheapening the convolutions of a network, coupled with a good distillation process, allows for a substantial reduction in the number of network parameters in return for a small drop in performance. However, the networks trained thus far have all had the same task – image classification. Here, we take an existing network, trained for the task of semantic segmentation and apply our method to distil it.

For our teacher network we use an ERFNet (Romera et al., 2017a,b) that has been trained *from scratch* on the Cityscapes dataset (Cordts et al., 2016) – a collection of images of urban street scenes, in which each pixel has been labelled as one of 19 classes. The bulk of an ERFNet is made up of standard residual blocks where each full convolution has been replaced by a pair of 1D alternatives: a 3×1 convolution followed by a 1×3 convolution. The second such pair in each block is often dilated. To *cheapen* this network for use as a student, we replace each block with a $G(N)$ block, maintaining the dilations where appropriate.

We use the same optimiser and training schedule as for the original ERFNet. When training the student, the only difference is the addition of an attention transfer term (see Equation 4.2) between several of the feature maps in the final loss. The models are evaluated using class Intersection-over-Union (IoU) accuracy on the validation set, and the results can be found in Table 4.5. Intersection-over-Union is a metric defined over pixel sets A and B:

$$\text{IoU}(A, B) = \frac{|A \cap B|}{|A \cup B|} \tag{4.5}$$

In Romera et al. (2017b), the authors detail how ERFNet is designed with efficiency in mind. With only one training run and no tuning, we are able to reduce the number of parameters to one quarter of the original for a modest drop in performance (2.06M to 0.49M parameters, 70.59 to 68.11 IoU).

Implementation Details Models were trained using the same optimiser, schedule, and image scaling and augmentation as in the ERFNet paper (Romera et al., 2017b), with the attention transfer loss for the case of ERFNet-G(N) as a student. For encoder training, the outputs of layers 7, 12, and 16 were used for attention transfer with $\beta = 1000$. For decoder training, the outputs of layers 19 and 22 were also used and β was dropped to 600 (so that the contribution of this term remains this same).

4.6 CONCLUSION

The variety of architectures that can be explored solely in deep learning on images is extremely large so we have focused here on a generic method to make an existing architecture more efficient. In doing so, we have also explored some simple ways architectures can be modified to have a reduced parameter count or multiply-add operations.

By using recent advances in distillation, we have shown that replacing blocks in the architecture with cheaper alternatives can yield very efficient networks, competing with the state of the art. In addition, the training process is not complicated, and reuses the hyper-parameters for the optimiser used to train the original network.

The architectures produced in the experiments in this chapter are competitive with the state of the art in compression. In Table 4.4 ResNet34-G(N) has performance competitive with CondenseNet-8 (Huang et al., 2017b), both using approximately 3 million parameters and 500 million mult-adds (although CondenseNet-8 uses only 300 million) to achieve 30% top-1 error, and is therefore at the state of the art in compression of ImageNet models. However, we have presented this as a generic compression strategy, applied to an older architecture (ResNet-32 (He et al., 2016a)). It demonstrates a practical route to state of the art efficiency that is easier to implement on any architecture that contributes to the field of efficient deep learning.

Image classification is a benchmark problem for deep neural networks. We have demonstrated that networks produced by a rote substitution of cheaper blocks into existing networks produces a range of architectures. This range allows a practitioner to tune the network they require according to the resources that are available. In addition we demonstrated that the same can be done for a semantic segmentation problem, reducing the size of ErfNet (Romera et al., 2017b) by 4 times with only a small reduction in performance.

Table 4.3: Student Network test error on CIFAR-10/100. Each network is a Wide ResNet with its depth-width (D-W) given in the first column and its block type (corresponding to Table 4.1 in the second. N refers to the channel width of each block and M refers to the channel width after the bottleneck where applicable. The total parameter cost of the networks for CIFAR-10 is given, as well as the number of Mult-Add operations they use. Note that CIFAR-100 networks use an extra 11.6K parameters and mult-adds over their CIFAR-10 equivalents as they have a larger linear classification layer. Errors are reported for (i) learning with no distillation i.e. from scratch (Scr), (ii) knowledge distillation with a teacher (KD), and attention transfer with a teacher (AT). The same teacher is used for training, and is given in the first row. This table shows that (i) through attention transfer it is possible to cut the number of parameters of a network, but retain high performance and (ii) for a similar number of parameters, students with cheap convolutional blocks outperform those with expensive convolutions and smaller architectures. Results on the Pareto boundaries in Figure 4.4 are shown in bold.

		CIFAR-10					CIFAR-100		
D-W	Block	Params (K)	MAdds (M)	Scr	KD	AT	Scr	KD	AT
T 40-2	S	2243.5	328.3	4.79	–	–	23.85	–	–
16-2	S	691.7	101.4	6.53	6.03	5.66	27.63	27.97	27.24
40-1	S	563.9	83.6	6.48	6.39	5.50	29.64	30.21	28.24
16-1	S	175.1	26.8	8.81	8.75	7.72	34.00	37.28	33.74
40-2	S-2x2	1007.1	147.4	5.89	6.03	5.09	27.20	26.98	26.09
40-2	**G(2)**	1359.0	198.1	5.30	**5.37**	4.87	25.94	24.92	24.45
40-2	**G(4)**	814.7	118.5	5.50	5.81	**5.00**	26.20	25.48	25.30
40-2	**G(8)**	542.5	78.7	5.92	5.72	**5.05**	26.49	26.64	25.71
40-2	**G(16)**	406.4	58.8	6.65	6.38	**5.13**	28.85	27.10	26.34
40-2	**G(N/16)**	641.3	133.9	5.72	5.72	**5.12**	27.08	26.11	25.78
40-2	**G(N/8)**	455.8	86.4	6.07	5.61	**5.06**	27.85	27.05	26.15
40-2	**G(N/4)**	363.1	62.6	6.93	6.45	**5.31**	28.91	27.93	26.85
40-2	G(N/2)	316.7	50.8	7.12	6.83	5.98	30.24	28.89	28.54
40-2	G(N)	293.5	44.8	8.51	8.01	6.57	31.84	29.99	30.06
40-2	B(2)	431.8	64.5	6.36	6.28	5.37	28.27	28.08	26.68
40-2	B(4)	150.9	22.8	7.94	7.83	6.93	31.63	33.63	30.56
40-2	**BG(2,2)**	286.7	43.3	6.12	6.25	**5.57**	28.51	28.82	28.28
40-2	**BG(2,4)**	214.1	32.7	6.75	6.75	**6.05**	29.39	29.25	28.54
40-2	**BG(2,8)**	177.8	27.3	6.94	6.98	**6.09**	30.21	29.34	28.89
40-2	**BG(2,16)**	159.7	24.7	6.77	6.97	**6.19**	30.57	30.54	29.46
40-2	BG(2,M/16)	238.3	46.8	6.26	6.50	6.02	29.69	28.69	29.05
40-2	**BG(2,M/8)**	189.9	34.4	6.75	6.49	**5.94**	29.09	29.13	28.16
40-2	**BG(2,M/4)**	165.7	28.2	7.06	7.15	**6.03**	30.42	30.28	28.60
40-2	**BG(2,M/2)**	153.6	25.1	7.45	7.47	**6.17**	30.44	30.66	29.51
40-2	**BG(2,M)**	147.6	23.6	7.95	7.99	**6.67**	30.90	31.18	30.03
40-2	**BG(4,M)**	81.4	13.0	9.04	8.61	**7.87**	33.64	37.34	32.89

Table 4.4: Top 1 and Top 5 classification errors (%) on the validation set of ImageNet for models (i) trained from scratch and (ii) those trained with attention transfer with ResNet-34 (Res34) as a teacher. Res18 refers to a ResNet-18, and Res18-0.5 is a ResNet-18 where the channel width in the last three groups is halved. Res34-G(x) is a ResNet-34 with each convolutional block replaced by a G(x) block. We can observe that for a particular parameter budget (3M or ~10M), the networks with cheap replacement blocks outperform those with reduced architectures. These trends follow for mult-adds. Note that the Res34 and Res18 scratch results were obtained from pre-trained PyTorch models.

			Scratch		AT	
Model	Params	Mult-Adds	Top 1	Top 5	Top 1	Top 5
Res34 **T**	21.8M	3.669G	26.73	8.57	–	–
Res18	11.7M	1.818G	30.36	11.02	29.18	10.05
Res34-G(4)	**8.1M**	**1.395G**	**26.61**	**8.62**	**26.58**	**8.43**
Res18-0.5	3.2M	909M	36.96	15.01	37.20	15.02
Res34-G(N)	**3.1M**	**559M**	**32.98**	**12.26**	**30.16**	**10.66**

Table 4.5: IoU accuracy (%) on the validation set of Cityscapes for (i) ERFNet and (ii) ERFNet with replacement blocks (ERFNet-G(N)). For ERFNet-G(N), the accuracy when trained from scratch (Scratch IoU) and when used as a student with the original ERFNet as a teacher (AT IoU) is given.

Model	Params	Mult-Adds	Scratch IoU	AT IoU
ERFNet	2.06M	3.73G	70.59	–
ERFNet-G(N)	**0.49M**	**1.19G**	**65.29**	**68.11**

5 COMPRESSED LINEAR TRANSFORMS

5.1 INTRODUCTION

Dense weight matrices store a floating point value at every index location, leading to a quadratic cost in both parameters and the operations to implement matrix-vector multiplications involving them. Here, we compare alternative linear transforms that can perform a function interchangeable with that performed by the original dense weight matrix, while using fewer parameters and/or fewer operations. As the use of dense weight matrices is ubiquitous in deep neural networks, this reduction could be of value in many settings.

Alternative linear transforms have been proposed in the literature, and typically have been demonstrated to improve performance when substituted in place of the large, fully connected final layers in AlexNet-like (Krizhevsky et al., 2012b) architectures, as detailed in Section 2.5.4. Since then, results have shown that these final layers are not as important as once thought. Hoffer et al. (2018) found that the final layer of deep image classification models need not be trained, and modern architectures, such as ResNets (He et al., 2016a), do not include massive fully connected final layers.

Using these substitutions in place of convolutions has been more difficult. HashedNet (Chen et al., 2015), a substitution we consider here, had to be substantially altered to produce useful compression in convolutional layers (Chen et al., 2016).

In this chapter we substitute a number of alternative linear transforms into *separable* convolutional layers (separable convolutions are described in Section 5.2.2) using each of these methods in place of the pointwise convolutional layer. Any method proposed as a replacement for a fully connected layer is also a replacement for a pointwise convolution because a pointwise convolution is a fully connected layer applied at all spatial points in the input tensor. Defining the convolution in this way also aids in implementing training routines for the proposed substitute linear transforms. In many cases, we can compute a substitute weight matrix, and differentiate back through that computation in training, which avoids memory expansion caused by storing activations at extra intermediate points.

The contributions made in this work are:

- State of the art compression factors, top-1 error versus either parameter or mult-add cost, using only *linear* transformations of existing neural network architectures.
- Demonstrating the practical applicability of a number of alternatives to the standard fully connected layers in modern large-scale image classification architectures for the first time.
- A comprehensive comparison of such alternative layers, with a detailed discussion of the comparative advantages for each approach.
- A basic derivation of a simple rule to stabilise the training of compressed linear transforms, along with ablation experiments to demonstrate the value of such a method, allowing these methods to be applied across a range of architectures of different sizes; results that have not been possible in any prior work.

In Section 5.2.1 we introduce the six methods we will compare. Section 5.2.2 contains the definition of a separable convolution and justifies their use in this work. A full description of each method can be found in Section 5.2.3. Experimental design to give the proposed methods a fair comparison is described in Section 5.2.4. Solutions to the difficulty of training networks with these substitutions are proposed in Section 5.2.5.

Deep neural networks have gained popularity in part due to the impressive results on image classification problems, such as those of AlexNet (Krizhevsky et al., 2012b). Yet, this is one area where these networks are resource intensive, requiring large memory and GPU. The goal of this research is to improve efficiency, so image classification is a natural task to build our experiments around. Choosing this task is justified by other research on efficiency, which has typically compared results on image classification problems, as described in Chapter 2.

In Section 5.3 we present comprehensive experiments on the CIFAR-10 (Krizhevsky, 2009) dataset, with both common and state of the art architectures, and on the ImageNet (Deng et al., 2009) dataset. We compare all proposed substitute convolutions in terms of their top-1 error with respect to the original network, and how this error changes as we vary the parameter budget or the number of mult-adds used. We also present ablation experiments in Sections 5.3.5 and 5.3.6 demonstrating that the proposed protocols in Section 5.2.5 are necessary to train networks using these substitute convolutions.

5.1.1 *Related Work*

Our approach can be compared to work yielding a low-rank tensor to use in the convolutional layers. Previous efforts on low-rank convolutional networks have focused on transforming pre-trained networks (Jaderberg et al., 2014; Alvarez and Petersson, 2016; Denton et al., 2014; Lebedev et al., 2014) or training networks with appropriate regularisers (Alvarez and Salzmann, 2017; Wen et al., 2017). A complete overview of compressed linear transforms can be found in Sections 2.5.4 and 2.5.5.

Networks with low-rank constraints are markedly more difficult to train. In Garipov et al. (2016) the authors train such a network on CIFAR-10, but only achieve a 2× compression rate over a convolutional network, and attain less than 90% accuracy. Other papers have focused on similar tensor decompositions; Su et al. (2018) obtain 91.28% accuracy compressing a ResNet-34. These decompositions can offer some speed increases, but we were unable to replicate the results in these papers. Also, the necessary algorithms to run fast matrix-vector products whilst between tensor-decomposed representations are an area of active research (Oseledets, 2011).

5.2 METHODS

Our experiments are designed to demonstrate that compressed convolutional layers can indeed be trained, and can learn to represent the linear transformations necessary for deep learning. We focus on comparing various substitutions for the linear transforms used in deep learning that either use fewer parameters than a full dense matrix, or fewer mult-adds, or both.

From the literature, we have selected a subset of the available methods, based on reported results and the prospect of incorporating the method in a generic training setting. We list these methods in Section 5.2.1.

In our experiments involving these methods we decided to use depthwise-separable convolutions. Description and justification for this decision can be found in Section 5.2.2. The details on how each method is then composed as a separable convolution are presented in Section 5.2.3.

Finally, we describe the techniques used to stabilise training in Section 5.2.5. We aim to show that these compressed layers *can learn* the functions necessary to implement a deep neural network. As such, it is important to observe the performance using model distillation, in order to achieve a result closer to the maximum potential of a compressed linear transform.

5.2.1 *Methods To Compare*

Candidate methods for matrix substitution were chosen from the literature. We selected work that is prominent and also allows a simple implementation in training. In the context of this thesis we do not focus on implementing the most efficient algorithm for a given method, focusing only on *whether the method can be trained*. We rely on the results from each original paper to indicate the efficiency of each method.

The following methods were selected and each is described in full in Section 5.2.3.

- ACDC (Moczulski et al., 2015) was chosen a simple effective candidate structured efficient linear layer, an overview of which is provided in Section 2.5.5.

- The Tensor-Train decomposition (Novikov et al., 2015; Garipov et al., 2016) has been demonstrated as an effective substitution for linear layers in neural networks, but work has not focused on the convolutional layers. An overview of this method can be found in Section 2.5.4.

- The Tucker decomposition, also known as the higher order singular value decomposition, provides an alternative method to decompose a higher dimensional tensor. By leveraging the smaller high-dimensional representation, it can offer significant computational efficiency benefits (Kossaifi et al., 2017).

- HashedNet (Chen et al., 2015) was used as a baseline to compare against by Novikov et al. (2015). As a compression method, it allows for any arbitrary compression ratio, with no effect on the execution time of the network. However, the method for compression does not use any arithmetic operations, unlike every other method considered, to build the resulting linear transform.

- ShuffleNet is a state-of-the-art efficient neural network architecture, and it achieves this with a particular design of convolutional block. Due to its empirical success, and the similarity between the combination of block-diagonal matrices and permutations of ShuffleNet and the structured efficient transforms of ACDC (Moczulski et al., 2015), it was chosen as a useful method to compare against.

Finally, we compare all of these to a linear bottleneck baseline, which we refer to as *Rank Factorized (RF)*, which is also described in Section 5.2.3.

5.2.2 *Separable Convolutions*

A depthwise-separable convolution implements a convolution with any kernel size by preceding a pointwise convolution with a grouped convolution, at the specified kernel size. The grouped convolution uses a number of groups equal to the number of input channels, with the effect of performing independent convolutions on slices of the input tensor one channel deep. This is known as a depthwise separable convolution and has been demonstrated as a substitute for convolution (Chollet, 2016). This type of convolution is now more common than conventional convolutions in state of the art networks, such as NASNet (Zoph et al., 2017).

In a depthwise separable convolution, the grouped spatial convolution typically has *far fewer parameters* than the pointwise convolution. In this work, we only substitute alternatives in place of the pointwise convolution.

Many of the methods here have only demonstrated results in place of the fully connected layers in deep neural networks, typically the final layers. A pointwise convolution can be seen as a fully connected layer, applied in parallel at every spatial position on the input tensor. We should expect methods that have been demonstrated to perform well as replacements for fully connected layers to work as substitutions for pointwise convolutions.

Another constraint that pushes us to choose only to focus on depthwise separable substitutions is the fact that structured efficient linear layers, those discussed in Section 2.5.5, and ACDC (Moczulski et al., 2015), always produce square weight matrices. Convolution is often implemented by first producing a *kernel matrix* where each column is a patch over the input space, allowing convolution to be implemented by matrix multiplication with filters aligned as rows in a weight matrix. This is known as the im2col-gemm algorithm. The kernel matrices applied in the im2col-gemm algorithm to implement full convolution are almost never square in deep neural networks.

Finally we note that one method reviewed, the linear ShuffleNet substitution, is different. It saves parameters by applying the grouped convolution in a bottleneck. This is described in more detail in the following section. We chose to keep this in line with the published work, despite the difference between competing methods.

5.2.3 *Substitute Linear Transformations*

In this section, we describe the composition of each of the methods being compared. All provide an approximation to the application of a dense random matrix in a linear

layer; a matrix-vector product of that matrix with an input vector. To be explicit, this is

$$\mathbf{y} = \mathbf{W}\mathbf{x}, \tag{5.1}$$

where $\mathbf{y}$ is the output vector, $\mathbf{W}$ is the dense random matrix, and $\mathbf{x}$ is the input vector.

Rank Factorized (RF) We have chosen a linear bottleneck transformation as a baseline against which to compare methods from the literature. In place of the dense random matrix in a linear transform, we first map an input to a smaller number of dimensions, and then back to the output number of dimensions. This uses two weight matrices $\mathbf{W}_1 \in \mathbb{R}^{d_{bn} \times d_{in}}$ and $\mathbf{W}_2 \in \mathbb{R}^{d_{out} \times d_{bn}}$, where the input dimensionality is d_{in}, bottleneck is d_{bn} and output is d_{out}. The linear transformation from an input $\mathbf{X}$ to an output $\mathbf{Y}$ can then be expressed:

$$\mathbf{y} = \mathbf{W}\mathbf{x} = \mathbf{W}_2(\mathbf{W}_1\mathbf{x}) = \mathbf{W}_2\mathbf{W}_1\mathbf{x} \tag{5.2}$$

We can recover the dense weight matrix by observing that $\mathbf{W} = \mathbf{W}_2\mathbf{W}_1$.

This parameterisation can be implemented in popular deep learning frameworks with two linear layers in sequence, but despite this simplicity it can give significant efficiency benefits. The number of parameters used by applying a dense weight matrix $\mathbf{W}$ to an input vector is $d_{out} \times d_{in}$, while the total parameters used in $\mathbf{W}_1$ and $\mathbf{W}_2$ is $d_{out} \times d_{bn} + d_{bn} \times d_{in}$.

For simplicity, if we assume $d_{out} = d_{in} = d$ and $d_{bn} = \frac{d}{b}$, then we can see the number of parameters used will be:

$$d_{out} \times d_{bn} + d_{bn} \times d_{in} = \frac{d^2}{b} + \frac{d^2}{b} = \frac{2d^2}{b} \tag{5.3}$$

The total parameters used is therefore $O(b^{-1})$, with a minimum at $b = d$ of of $2d$ parameters. Also, if we assume a matrix-vector product using these matrices is applied using a naive implementation according to the definition of matrix multiplication (as opposed to a common efficient implementation such as the Coppersmith-Winograd algorithm (Coppersmith and Winograd, 1990)) then this parameter count is also exactly equal to the number of mult-add operations used. However, in practice we are trading two smaller sequential matrix operations for one larger, which may be slower on some hardware as we have increased the number of sequential operations.

ACDC This parameterisation uses diagonal matrices $\mathbf{A}$ and $\mathbf{D}$, forward and inverse discrete cosine transforms (DCT) $\mathbf{C}$ and $\mathbf{C}^{-1}$ and permutation matrix $\mathbf{P}$. These are repeated L times in the following composition:

$$\mathbf{W} = \prod_{l=1}^{L} \mathbf{A}_l \mathbf{C} \mathbf{D}_l \mathbf{C}^{-1} \mathbf{P}. \tag{5.4}$$

Using the weight matrix in Equation 5.4 is equivalent to a stack of ACDC layers (Moczulski et al., 2015). Each ACDC layer being composed of a sequence of operations described by matrices in Equation 5.4, but applicable in fewer operations than the matrix multiplication. For $\mathbf{W} \in \mathbb{R}^{N \times N}$ the computational complexity is $O(N \log N)$ and storage cost is $2N$, as we can see by breaking down the process of multiplication with an input vector of size N:

1. Application of diagonal matrix $\mathbf{A}_l$, using N operations and costing N parameters.
2. A forward DCT, denoted by the DCT matrix $\mathbf{C}$; complexity $O(N \log N)$.
3. Application of diagonal matrix $\mathbf{D}_l$, using N operations and costing N parameters.
4. An inverse DCT, denoted by the inverse DCT matrix $\mathbf{C}^{-1}$; complexity $O(N \log N)$.
5. A random permutation, denoted by permutation matrix $\mathbf{P}$, using only memory indexing, but this can be time consuming in practice, so we use a *riffle shuffle*.

A *riffle shuffle* is a fixed permutation, splitting the input in half and then interleaving the two halves; equivalent to a perfect riffle shuffle with a deck of cards (Gilbert, 1955). This was found to work as well as a fixed random permutation and can be evaluated much faster as observed by Zhang et al. (2017a).

Linear Approximation We compared the riffle shuffle to a fixed random permutation on the toy synthetic regression problem described in Section 6.1 of Moczulski et al. (2015). Both random permutations and riffle shuffles converged to a final mean squared error of 0.02.

Substituting this parameterisation into convolutional layers presents a problem: most kernel matrices are not square, but all the component matrices here are, including diagonal, DCT and permutation matrices. This is one reason we focus on substituting only pointwise convolutions. Kernel matrices in pointwise convolutions are square when the number of input channels matches the output, which is true for

the majority of layers in the deep neural networks tested. To increase the number of channels, we repeat the input along the channel dimension. As channels commonly increase in integer steps, this allows us to implement all the pointwise convolutions we required.

Unfortunately, it is not practical to train a network using the sequence of component operations. A naive implementation in an automatic differentiation system will store a full activation tensor at every stage. The memory cost then grows with $5L \times S$, where S is the original storage cost of the activation tensor a traditional convolution would use. To avoid this, at training time, we compute $\mathbf{W}$ using Equation 5.4 and substitute it for use in the pointwise convolution.

At test time, the activations no longer need to be stored, and we are free to use an efficient implementation of the DCT to implement the convolutions. One ACDC layer is estimated to use $4N + 5N\log_2(N)$ mult-adds, and we use this to calculate the number of mult-adds the network uses at test time (Moczulski et al., 2015).

HashedNet A virtual weight matrix $\mathbf{V}$ is built from fewer real weights $\mathbf{w}$ using a hash function $\mathbf{h}$ to index those weights:

$$y_i = \sum_{j=1}^{N} V_{ij} x_j = \sum_{j=1}^{N} w_{h(i,j)} x_j. \tag{5.5}$$

Hash functions are often used for fast retrieval of a object in computing. HashedNets use a hash function to retrieve the weights used in their network (Chen et al., 2015). The particular hash function used in this case takes as input indexes in the "virtual" weight matrix, $\mathbf{V}$, used in the linear transformation, and produces as output a single index into a set of "real" weights $\mathbf{w}$. Figure 5.1 shows virtual weight matrices V^1 and V^2 being produced by hash indexing of real weights w^1 and w^2. These weight matrices are then applied in place of stored weight matrices in the network.

The indexes produced by the hash function are approximately uniform over the set of real weights. This produces a weight matrix in which weights are randomly tied, with each unique weight occurring on average the same number of times. Chen et al. (2015) demonstrate that the cost of accessing these weights is negligible at test time. In our experiments, we do not use a hash function, instead sampling the indexes once when the layer is initialised and storing them.

The number of parameters to be optimized in this case is the number of "real" weights $\mathbf{w}$, which can be set to be 1 or greater, up to the number of elements in the virtual weight matrix. However, as the number of real weights is increased the probability we may store a weight that is never used in the virtual weight matrix increases. If N_r is the number of real weights and N_v is the number of virtual weights,

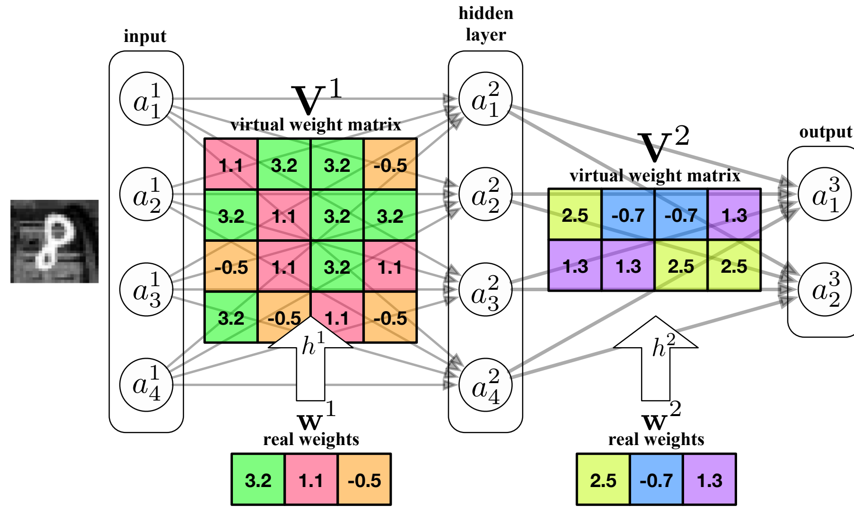


Figure 5.1: "An illustration of a neural network with random weight sharing under compression factor $\frac{1}{4}$. The $16+9=24$ virtual weights are compressed into 6 real weights. The colors represent matrix elements that share the same weight value." (Chen et al., 2015)

then the expected number of weights that will be excluded will be $N_r(1-1/N_r)^{N_v}$. Defining N_r in terms of N_v using a compression ratio $c = \frac{N_r}{N_v}$, we can investigate what happens to the ratio excluded, η, as c changes:

$$\eta = \left(1 - \frac{1}{cN_v}\right)^{N_v}. \tag{5.6}$$

Taking the limit in the case of large N_v, we can see this limit has the functional form of the limit definition of an exponential, $\exp(x) = \lim_{n\to\inf}\left(1+\frac{x}{N}\right)$:

$$\eta = \lim_{N_v\to\inf}\left(1 - \frac{1}{cN_v}\right)^{N_v} = \exp\left(-\frac{1}{c}\right) \tag{5.7}$$

As shown in Figure 5.2, this limit argument holds true for the values of N_v we are interested in, and the proportion of weights excluded as the compression ratio grows can be significant. In our experiments we do not address these wasted parameters, despite performing experiments with compression ratios in regions where 10-20% of our parameters are being excluded. It would also be possible to identify these parameters and choose not to store them, but we do not investigate this.

One reason we do not investigate this is that we find the HashedNet substitution effective at high compression levels, such as below $c = 0.1$, and in this region a negligible number of weights will be excluded. Excluded weights would be stored for no reason, so it is preferable to avoid them.

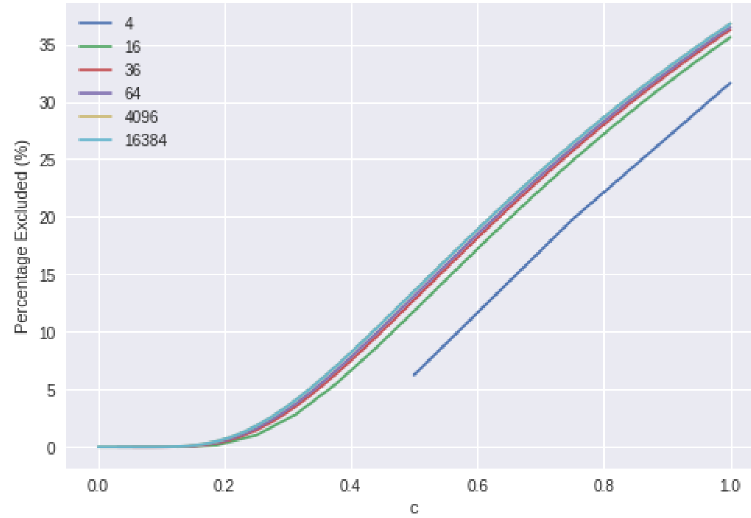


Figure 5.2: The effect on percentage of weights excluded depending on compression ration c, tested for different values of N_v, the number of elements in the virtual weight matrix, indicated in the legend. At the compression levels we are interested in, 20% of the original number of weights, we can see that the number of weights excluded is low.

Tensor-Train Decomposition The weights in a convolutional layer are typically stored in a 4D tensor, and in a linear layer in a 2D weight matrix. Assuming we have some higher dimensional tensor we assume it is possible to map this tensor to our weight matrix using a reshape operation:

$$\mathbf{y} = \mathbf{W}\mathbf{x} = \text{reshape}^{\mathbb{R} \in N \times N}(\boldsymbol{\mathcal{A}})\mathbf{x}. \tag{5.8}$$

We can use a tensor decomposition to represent $\boldsymbol{\mathcal{A}}$ and implement the linear transform using fewer floating point parameters. For example, the canonical decomposition of this d-dimensional tensor can be expressed as:

$$\boldsymbol{\mathcal{A}}(i_1, ..., i_d) = \sum_{\alpha=1}^{r} \mathbf{U}_1(i_1, \alpha)...\mathbf{U}_d(i_d, \alpha) \tag{5.9}$$

Where r is the canonical rank of the tensor, and the U_k matrices are known as canonical factors (Oseledets, 2011). Unfortunately, Oseledets (2011) notes that the algorithms for finding this decomposition, even within a given error, are not reliable. As an alternative they propose the Tensor-Train (TT) decomposition:

$$\boldsymbol{\mathcal{A}}(i_i, ..., i_d) = \mathbf{G}_1(i_1)...\mathbf{G}(i_d) \tag{5.10}$$

Where $\mathbf{G}_k(i_k)$ are $r_{k-1} \times r_k$ matrices, with the boundary conditions that $r_0 = r_d = 1$. Each element of the tensor can then be reproduced by performing this sequence of matrix products. However, it is possible to perform a matrix-vector, or matrix-matrix, product between two TT tensors without having to map back to a 2D matrix at any point. Due to time constraints and unavailability of a reference implementation, in our experiments we compute the weight matrix from the $\mathbf{G}_k$ factors and backpropagate the error to update those factors with automatic differentiation.

The parameter savings using this method depend on the number of dimensions possessed by the tensor storing the weights. In our experiments we found it best to reshape weight matrices to 3 dimensions, with approximately equal sizes. We then set the TT-rank $r_1, ..., r_{d-1}$ to control the level of compression. This is in line with previous work substituting TT tensors into deep neural networks for compression, such as the work of Novikov et al. (2015).

It is worth noting that TT tensor decompositions can be better adapted in deep learning. Garipov et al. (2016) investigated an alternative way to parameterize convolutional layers that they found more effective. We focus here on the simpler version, to verify that a TT tensor can store the functions necessary in a deep network.

A full TT tensor deep neural network, with activations stored in high dimensional tensors, has yet to be demonstrated in the literature. The matrix-vector product can be much more efficiently performed with TT tensors, rather than conventionally, but it causes growth of the TT-rank in the resulting tensor (Oseledets, 2011). Therefore, algorithms for efficient matrix-vector products on TT-tensors have to combine the multiplication with a TT-rounding step to avoid continual growth, which can become $O(dn^2 r_k^6)$, where n is the size of any dimension.

In our experiments, we do not calculate how efficient an implementation using TT tensors could be. However, research on efficient TT matrix-vector products is ongoing, so if we can demonstrate that TT tensors can store the parameters required for a deep neural network to operate, it will help to yield a faster way to implement the linear transformations in such networks.

Tucker This method also uses a tensor decomposition to define $\mathbf{W}$ as described in Equation 5.8. The Tucker decomposition again decomposes a tensor $\boldsymbol{\mathcal{A}} \in \mathbb{R}^{I_0, ..., I_d}$, but in this case uses a low rank core $\boldsymbol{\mathcal{G}} \in \mathbb{R}^{R_0, ..., R_d}$ projected by factors $\mathbf{U}_k \in \mathbb{R}^{R_k, I_k}$ (Kossaifi et al., 2017):

$$\boldsymbol{\mathcal{A}} = \boldsymbol{\mathcal{G}} \times_0 \mathbf{U}_0 ... \times_d \mathbf{U}_d. \tag{5.11}$$

The parameter cost of this decomposition scales exponentially with d; much faster than TT, which is linear. However, Kossaifi et al. (2017) used this transformation to define a "tensor contraction layer", which, along with a "tensor regression layer" was successful in matching the performance of traditional networks while using far fewer parameters.

In our experiments, to compare with TT, we only use the Tucker decomposition to store our weight matrices. As with the TT decomposition, we compute the weight matrix, then backpropagate gradients in order to update the $\mathbf{U}_k$ factors.

ShuffleNet A ShuffleNet block is composed of a grouped pointwise convolution, a channel shuffle operation, a 3×3 depthwise separable convolution and a final grouped pointwise convolution. We implement it with all of these components, in this order, without nonlinearities. For comparison, if we ignore the 3×3 depthwise separable convolution, this linear transform can be expressed as:

$$\mathbf{y} = \mathbf{W}\mathbf{x} = \mathbf{B}_2\mathbf{P}\mathbf{B}_1\mathbf{x} \tag{5.12}$$

Where $\mathbf{B}_1$ and $\mathbf{B}_2$ are block diagonal matrices implemented by grouped 1×1 convolutions, and $\mathbf{P}$ is a permutation implemented by a riffle shuffle. Including the depthwise convolution, with its weight tensor $\boldsymbol{\mathcal{D}}$, we can express the whole tensor operation with a convolution operator $c(\text{input}, \text{weight})$:

$$\boldsymbol{\mathcal{Y}} = c(c(\text{riffle}(c(\boldsymbol{\mathcal{X}}, \boldsymbol{\mathcal{B}}_1)), \boldsymbol{\mathcal{D}}), \boldsymbol{\mathcal{B}}_2) \tag{5.13}$$

Where $\boldsymbol{\mathcal{Y}}$ and $\boldsymbol{\mathcal{X}}$ are input and output tensors, $\boldsymbol{\mathcal{B}}_1$ and $\boldsymbol{\mathcal{B}}_2$ are defined by adding dimensions of size 1 to $\mathbf{B}_1$ and $\mathbf{B}_2$: $\mathbb{R}^{N \times N} \to \mathbb{R}^{N \times N \times 1 \times 1}$. The permutation $\mathbf{P}$ here is referred to as riffle, denoting the channel-wise riffle shuffle.

While this was not proposed in the literature as a method to compress a linear transformation, the building blocks involved are similar to those used in the ACDC structured efficient linear transformation. In place of the diagonal matrices, DCT and permutations, it is composed of block-diagonal matrices and permutations.

Zhang et al. (2017a) found that the permutation could be efficiently implemented using a riffle shuffle. Despite not implementing a true random permutation, they found that the interconnection this provided between convolutional groups was sufficient to achieve good performance. In Figure 5.3 the blocks used in their paper are illustrated.

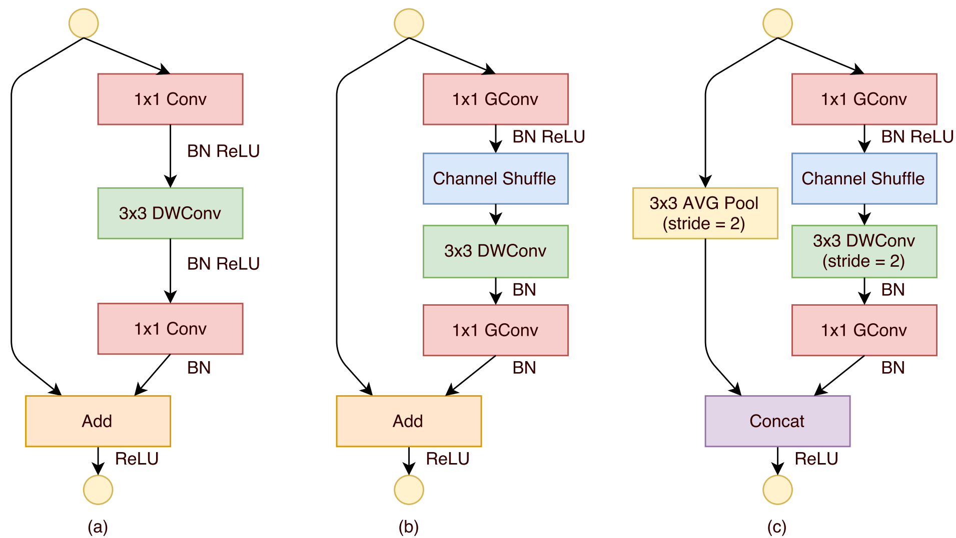


Figure 5.3: "ShuffleNet Units. a) bottleneck unit (He et al., 2016a) with depthwise convolution (DWConv) (Chollet, 2016; Howard et al., 2017); b) ShuffleNet unit with pointwise group convolution (GConv) and channel shuffle; c) ShuffleNet unit with stride = 2." (Zhang et al., 2017a)

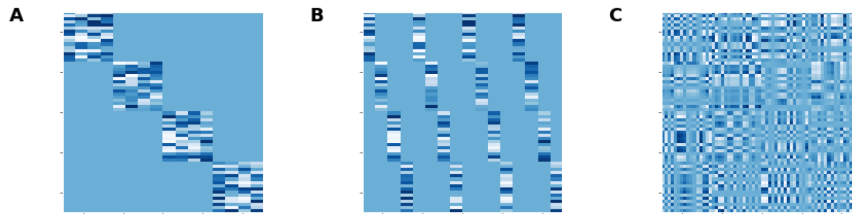


Figure 5.4: The first GConv in Figure 5.3 is illustrated in *A*. The proceeding channel shuffle connects these independent groups in *B*. Passing through the final GConv block produces the approximately dense random matrix in *C*.

In this work, we are comparing linear transformations, so we propose a linear version of the transformations in Figure 5.3. This consists of the components in Figure 5.3 b), removing the nonlinearity, batchnorm and skip connection. Figure 5.4 demonstrates how this process builds up a matrix resembling a dense random matrix by allowing cross-connections between channel groups.

The resultant kernel matrix is illustrated in Figure 5.4.

Unlike every other method compared, the depthwise-separable convolution, with kernel size 3, is applied *in a bottleneck* between the two grouped pointwise convolutions. The number of channels in this bottleneck is always exactly the number of input channels divided by 4.

5.2.4 *Parameter Cost Grid Search*

All of the methods tested have a single tuning parameter allowing us to vary the number of parameters used by a model. As we have three different models in the different experiments, we have to tune these parameters for each. In each case, we tune the number of parameters to be approximately equal regardless of the method of substitution. These scaling factors are described in the previous Section.

We perform experiments on CIFAR-10 (Krizhevsky, 2009) with WRN-28-10 (Zagoruyko and Komodakis, 2016) and DARTS (Liu et al., 2018) and on ImageNet with WRN-50-2. Technical details on the implementation of these experiments is given in Section 5.3.1. We look at the following parameter budgets for each:

- *WRN-28-10*: 2.38M, 1.2M, 0.6M
- *DARTS*: 1.42M, 0.83M, 0.49M
- *WRN-50-2*: 17.7M, 4.35M

After normalising the tuning of all layers between 0 and 1, we can plot number of parameters used by each substitution as shown in Figure 5.5. Different compression methods produce a curve of model sizes depending on the tuning setting; most increase over the tuning parameter range, apart from the number of groups used by linearised ShuffleNet. The upper limit and lower limits were chosen where all methods have support. For example, we can see in Figure 5.5 we can see that the upper limit is defined by the Linear ShuffleNet, while the lower limit is defined by RF. We chose the midpoint by linear interpolation in log parameter count.

5.2.5 *Training Compressed Linear Transforms*

There are two factors enabling us to adequately test the performance possible in deep neural networks when using the proposed substitute layers. First, we use model distillation, specifically attention transfer (Zagoruyko and Komodakis, 2017) (AT) to stabilise training. Second, we find that training in these networks could be sensitive to the weight decay used on the parameters used in these compressed layers.

One way to motivate L2 regularisation in neural networks is to say that it is equivalent to MAP inference with a normal prior on the weights (Murphy, 2012, p.225). We observed in early experiments that the results obtained with ACDC substitutions were dependent on the value of the weight decay. When we replace a weight matrix with a compressed version, expressing the original with fewer parameters, we might

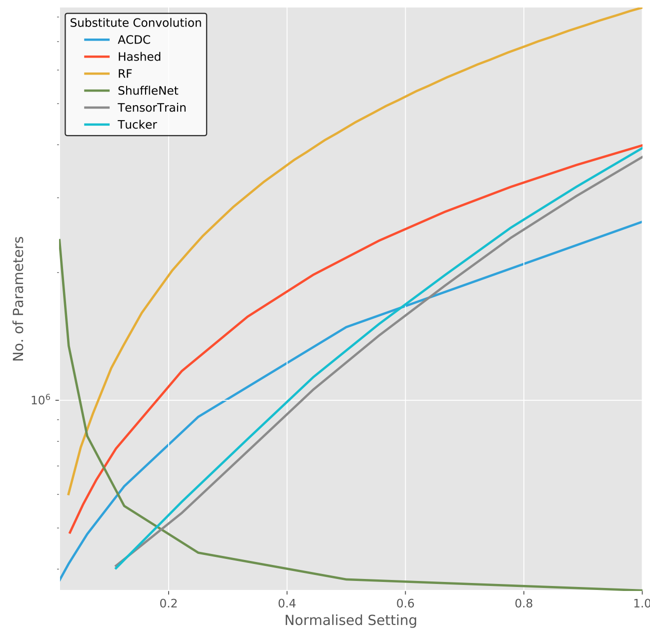


Figure 5.5: The parameter cost of a WRN-28-10 after substitution by the methods listed in the legend, varying the tunable parameter of each over a normalised range. We design experiments over a parameter count range such that all methods illustrated will have support, which here is limited by the maximum size of the Linear ShuffleNet and the minimum size of the RF substitution.

expect that we ought to use a different weight decay factor. After all, we have fewer parameters, so we can accept fewer of them dropping to zero due to weight decay. The question then is, by how much should that weight decay factor be reduced?

A rule of thumb we could appeal to would be to preserve the total variance of the weight matrix prior. As we will see, this has a useful property: as the number of parameters tends towards the number in the full weight matrix, we will tend toward the original weight decay factor. If we assumed the weights, $\{w_n\}_{n=1}^{N}$ are normally distributed with variance equal to $\frac{1}{\sqrt{d}}$, where d is the weight decay factor, then total variance is

$$\sum_{n=1}^{N} \mathrm{E}\left[w_n^2\right] = \sum_{n=1}^{N} \frac{1}{d} \mathrm{E}\left[z^2\right], \tag{5.14}$$

where we have employed the reparameterization trick, and z is normally distributed with variance 1. We have

$$\sum_{n=1}^{N} \mathrm{Var}(w_n) = \frac{N}{d}. \tag{5.15}$$

If we assume we have real parameters in our compressed version of the weight matrix $\{\theta_m\}_{m=1}^{M}$, also normally distributed with variance $\frac{1}{\sqrt{d_c}}$, then we can solve for d_c, the weight decay factor to use on our compressed weight matrix:

$$\sum_{n=1}^{N} Var(w_n) = \sum_{m=1}^{M} Var(\theta_m) \tag{5.16}$$

$$= \frac{N}{d} = \frac{M}{d_c} \tag{5.17}$$

$$= d_c = \frac{Md}{N}. \tag{5.18}$$

In practice this means multiplying the weight decay factor for compressed weight matrices by the compression ratio M/N. In Figure 5.13, it is illustrated that this indeed stabilises training and improves performance, and has the desirable property of providing a smooth interpolation to an uncompressed matrix – where the weight decay would simply return to the default. This way to set the weight decay will be referred to as *compression ratio scaled* (CRS) weight decay.

5.3 EXPERIMENTS

The comparison experiments comprise a grid search over parameter budgets on CIFAR-10 and ImageNet. A description of this grid search can be found in Section 5.2.4. The relationship between the performance of each substitution to the tasks and the number of parameters it uses is analysed in Section 5.3.3, and a similar analysis in terms of the mult-adds used by each substitution is given in Section 5.3.4.

Technical details on the design of experiments are given in Section 5.3.1. Settings used in the Tensor-Train and Tucker substitutions in later experiments are justified by experiment in Section 5.3.2. By ablation, the CRS weight decay used in these experiments is justified by experiment in Section 5.3.5, and the use of AT for distillation is justified in Section 5.3.6.

All experiments were written in Python using PyTorch (Paszke et al., 2017). Tensor-Train and Tucker decompositions were implemented using tntorch (Ballester, 2019); all other methods were implemented separately[1]. Experiments using the ImageNet dataset (Deng et al., 2009) were partly run using Amazon cloud credits for research. Figures were produced using Matplotlib (Hunter, 2007) and Holoviews (Stevens et al., 2015). Annotations on figures were placed using adjustText (Flyamer et al., 2018).

[1] The code implementing ACDC layers is based on the work of Hu (2019), and publicly available: https://github.com/gngdb/pytorch-acdc

5.3.1 *Experiment Setup*

Experiments on CIFAR-10 (Krizhevsky, 2009) were completed first and informed experiments on ImageNet (Deng et al., 2009). We were able to run hundreds of experiments on CIFAR-10, using both simple and complex networks and only 10 experiments on ImageNet.

CIFAR-10 is a set of 60,000 colour images of size 32 by 32 pixels, with the task of classifying each image according to 10 classes (Krizhevsky, 2009). ImageNet is a dataset of 1 million colour images of size 224 by 224, with the task of classifying each into 1000 classes (Deng et al., 2009). The results on CIFAR-10 typically inform experiments planned on ImageNet, which is used as verification that the method scales to large problems.

The shortest experiments were run using a codebase called cifar10-fast (Page, 2019), allowing networks to be trained around 10 times faster than later experiments. These experiments were used to decide on settings to use in longer running experiments producing the results upon which we base our arguments. These can be seen in Figures 5.6.

Each network was trained for 128 epochs with a cosine annealed schedule starting at 0.2, with Nesterov momentum (Sutskever et al., 2013) set to 0.9 and a minibatch size of 512. The original weight decay setting was 5×10^{-4}, modified only when testing alternative settings. The data was augmented with random crops, left-right flips and Cutout (Devries and Taylor, 2017)[2].

The next round of CIFAR-10 experiments were designed to match the experiments performed in the papers introducing the network architecture. We focused on two architectures: Wide ResNets (Zagoruyko and Komodakis, 2016) (WRN) and the network found in Differentiable Architecture Search (Liu et al., 2018) (DARTS). Wide ResNets were chosen to demonstrate results on a common ResNet structure. Results on this type of network should be reflected in many similar networks in the literature. Wide ResNets are defined by their *depth* and *width* factors. We choose to focus on the WRN-28-10, depth $= 28$ and width $= 10$, which is the largest network, with the lowest top-1 error, considered by Zagoruyko and Komodakis (2016).

Wide ResNets architectures were used to demonstrate the results of attention transfer (Zagoruyko and Komodakis, 2017), and we run these networks using that training protocol. When using attention transfer α was set to 0 and β was set to 1000.

---

[2]The full source code to run these experiments is publicly available: https://github.com/gngdb/cifar10-fast.

DARTS was selected in order to demonstrate results on a state-of-the-art image classification architecture. We replicated precisely the training hyperparameters and schedule used in the original paper. [3]

Wide ResNet Each network was trained for 200 epochs with a learning rate starting at 0.1 and scaled by 0.2 on epochs 60, 120 and 160. Momentum was set to 0.9 and the minibatch size was 128. Weight decay was set to 5×10^{-4} and scaled in all experiments according to the method described in Section 5.2.5, apart from the ablation experiment described in Section 5.3.5. Data was augmented with random crops, left-right flips and Cutout (Devries and Taylor, 2017).

DARTS Each network was trained for 600 epochs using a cosine annealed learning rate schedule starting at 0.025. Momentum was set to 0.9 and the minibatch size was 96. Weight decay was set to 3×10^{-4} and scaled in all experiments according to the method described in Section 5.2.5. The auxiliary classification head was used in training, but not counted at test time, and the drop-path method from the paper followed the same schedule of a linear increase in drop probability from 0 to 0.2 over the learning schedule. Data was augmented with random crops, left-right flips and Cutout (Devries and Taylor, 2017).

In ImageNet experiments we focused on a large network with competitive results, in order to demonstrate the potential for compression. As with previous experiments we chose a Wide ResNet (Zagoruyko and Komodakis, 2016), so that results could be interpreted as transferable to other ResNet-like architectures in the literature. All ImageNet experiments use the WRN-50-2, which is precisely a ResNet-50 (He et al., 2016a) with twice as many channels on inner bottlenecks. The published performance of 21.9% top-1 error is competitive with the best published results on ImageNet. We used the publicly available model zoo trained weights (Paszke et al., 2019) and found it could only achieve 22.5%. We chose to continue with this architecture as this performance is still comparable with the state of the art, and within 1% the published value.

Each network was trained for 90 epochs with a learning rate of 0.1 scaled by 0.1 at epochs 30 and 60. Momentum was set to 0.9 and the minibatch size was 256. Weight decay was 1×10^{-4} and scaled according to the method described in Section 5.2.5. Data was augmented with random crops and left-right flips.

Not all methods were compared in ImageNet experiments due to resource constraints. Each experiment took from several days to a week, depending on the method

[3] The code to run both sets of experiments is publicly available: `https://github.com/BayesWatch/deficient-efficient`.

and GPUs it was running on. Along with the RF method as a baseline, the methods that were run were those that were seen to perform best in the CIFAR-10 experiments: Tensor-Train, ShuffleNet and HashedNet.

5.3.2 *Tensor Decomposition Settings*

Tensor-Train and Tucker decompositions have previously been implemented for use in deep neural networks, as discussed in Section 5.2.3. We focus on replicating this method while ensuring we can control the compression of each layer with a single hyperparameter. Unfortunately, this requires that we choose the dimension of the tensor $\mathcal{A}$ we are decomposing.

Additionally, we could choose the rank of each TT-core, or each U factor in a Tucker decomposition, individually. To reduce the choice in the problem, we choose the size of TT-cores and U factors as a scaling factor times the size of the corresponding dimension. To change the level of compression, we only need to tune the scaling factor.

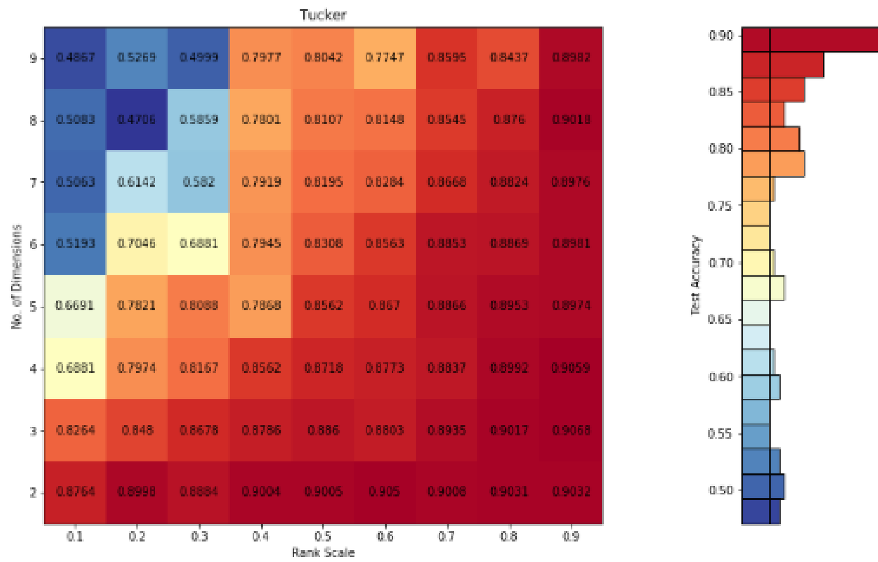
We only had to decide how many dimensions in which to represent the tensor. In order to decide this we performed a grid search over the number of dimensions and the rank scaling factor used. The results of this grid search, for TT and Tucker decompositions, are illustrated in Figure 5.6a and 5.6c. The acceptable top-1 error for both Tensor-Train and Tucker decompositions mostly only occurs at 4D and below.

Figure 5.6b and 5.6d show the compression ratio of the networks over this same grid search. To allow us to access a wide range of compression ratios with a minimal effect on the top-1 error achieved we chose to use 3D tensors in future experiments.

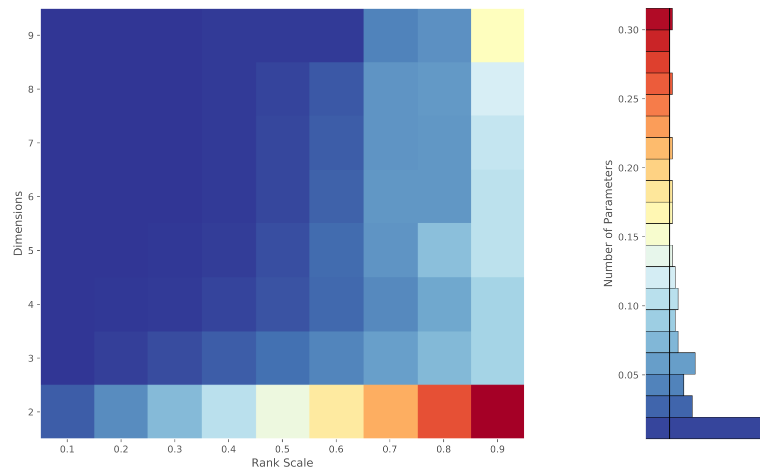
5.3.3 *Parameter Use*

Using a substitute compressed linear transform, we can reduce the number of parameters used by a deep neural network. In this section we compare, over the networks considered, what the effect is on the performance of a network when substituting our proposed set of linear transforms. As described in Section 5.3.1, this will be WRN-28-10 and DARTS on CIFAR-10, and WRN-50-2 on ImageNet.

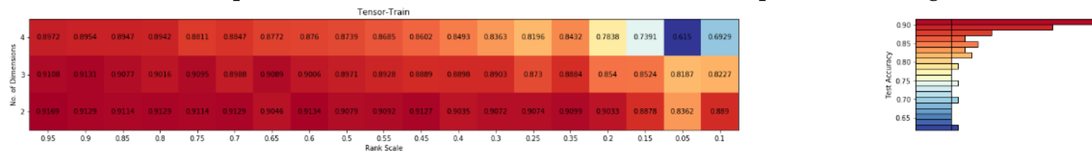
The following experiments relied on the results of WRN-28-10, so these will be presented first. In Figure 5.7 the relationship between the number of parameters used by a network and the top-1 error is illustrated. All AT results use a teacher network that is also the base network for substitution. It achieves a top-1 error of 3.2% and has 36.5M parameters. Without AT, it can be seen that the RF baseline substitution is



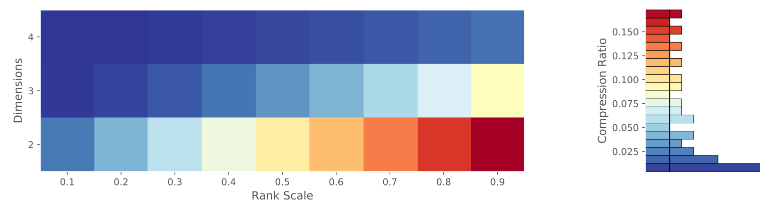
(a) Top-1 validation error as a function of Tucker decomposition settings.



(b) Compression ratio as a function of Tucker decomposition settings.



(c) Top-1 validation error as a function of Tensor-Train decomposition settings.



(d) Compression ratio as a function of Tensor-Train settings.

Figure 5.6: Results of a grid search over settings when using Tensor-Train and Tucker decomposition substitutions in a ResNet-18-like architecture optimised for fast experiments, as described in Section 5.3.1. Figures c and a show the relative top-1 error achieved over the gride search settings. Figures d and b show the variation in compression ratio over the same range. We can see that, while the compression ratio is higher as the dimensionality of the tensor increases, the top-1 error possible begins to suffer.

competitive with every other method. This may be surprising, given that it is simpler than some of the other methods listed here.

Once AT is enabled, we see that the methods we might expect to perform best, HashedNet and Tensor-Train, do have a small advantage, but only at lower compression ratios. One reason that we might expect HashedNet or Tensor-Train to work better as compression methods is that they do not necessarily reduce the number of mult-adds used by the network. HashedNet, in particular, substitutes a weight matrix of precisely the same size at test time, and applying that weight matrix uses the same number of mult-adds used by the original network.

AT always provides an advantage in training, as described in Section 5.3.6. In these experiments, we are only interested in whether these simplified linear transform *can* learn to perform the operations of dense random matrices, so we choose to use AT in all following experiments.

DARTS is a state-of-the-art network, as noted in Section 5.3.1, and the base network we substitute into achieves 2.83% error while using only 3.8×10^6 parameters. One reason it uses fewer parameters in comparison to WRN-28-10 is that it already uses separable convolutions. For this reason, we should not expect to see the same level of compression possible on WRN-28-10.

The results of a range of experiments using AT with the linear transforms substituted into DARTS are illustrated in Figure 5.8. At low compression the RF substitution performs similarly to other methods but as the level of compression increases the RF block finds a top-1 error 2% worse (higher) than competing methods. However, this may be more indicative of the high performance of competing substitutions.

Substituting into the DARTS network, while still being within 1% the original top-1 error, we could achieve compression to 20% of the original number of parameters for HashedNet, ShuffleNet and Tensor-Train substitutions (compared to 10% for 1% top-1 error tolerance with WRN-28-10). Also, if we look at these results in the lower plot of Figure 5.8 then we see that this conveniently explores an empty region of the Pareto frontier in the context set by the literature. The top-1 error achieved by HashedNet substitution is equal to or lower than all published networks compared against, save for DARTS and NASNet-A, while using several times fewer parameters. We also see the ShuffleNet substitution perform only marginally worse, while at the same time using around 5 times fewer mult-adds than the original network, or the HashedNet substitution, as illustrated in Figure 5.12.

Due to limited resources we decided it was not practical to run all the proposed linear transform substitutions on ImageNet, as noted in Section 5.3.1. Based on their performance in the two CIFAR-10 experiments, we chose HashedNet, Tensor-Train

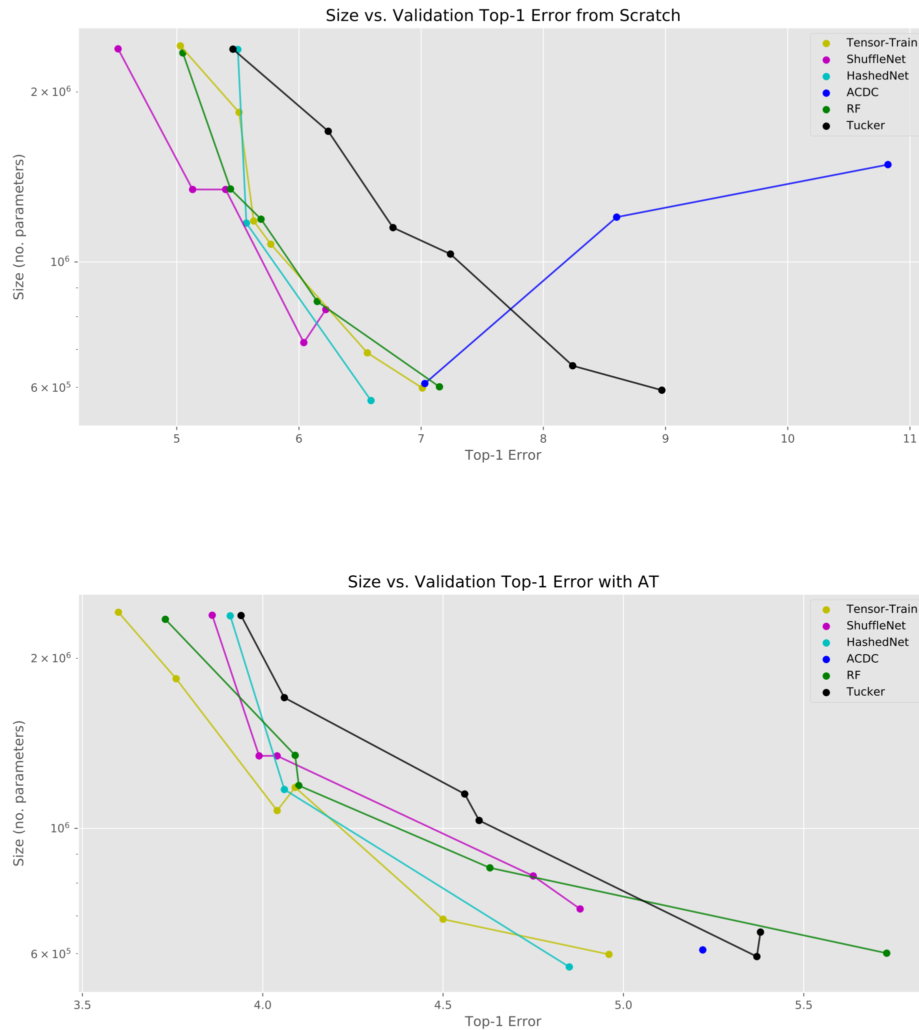


Figure 5.7: The relationship between top-1 error on the validation set and the number of parameters is plotted for experiments involving WRN-28-10 on CIFAR-10, for experiments using AT and without. Each substitute linear transform tested is indicated in the legend. On this problem, both Tensor-Train and HashedNet substitutions are able to achieve the highest rates of compression. At lower compression settings, all methods compared achieve comparable top-1 error. Note that ACDC was unstable for larger networks and so we only plot results for the smallest parameter budget.

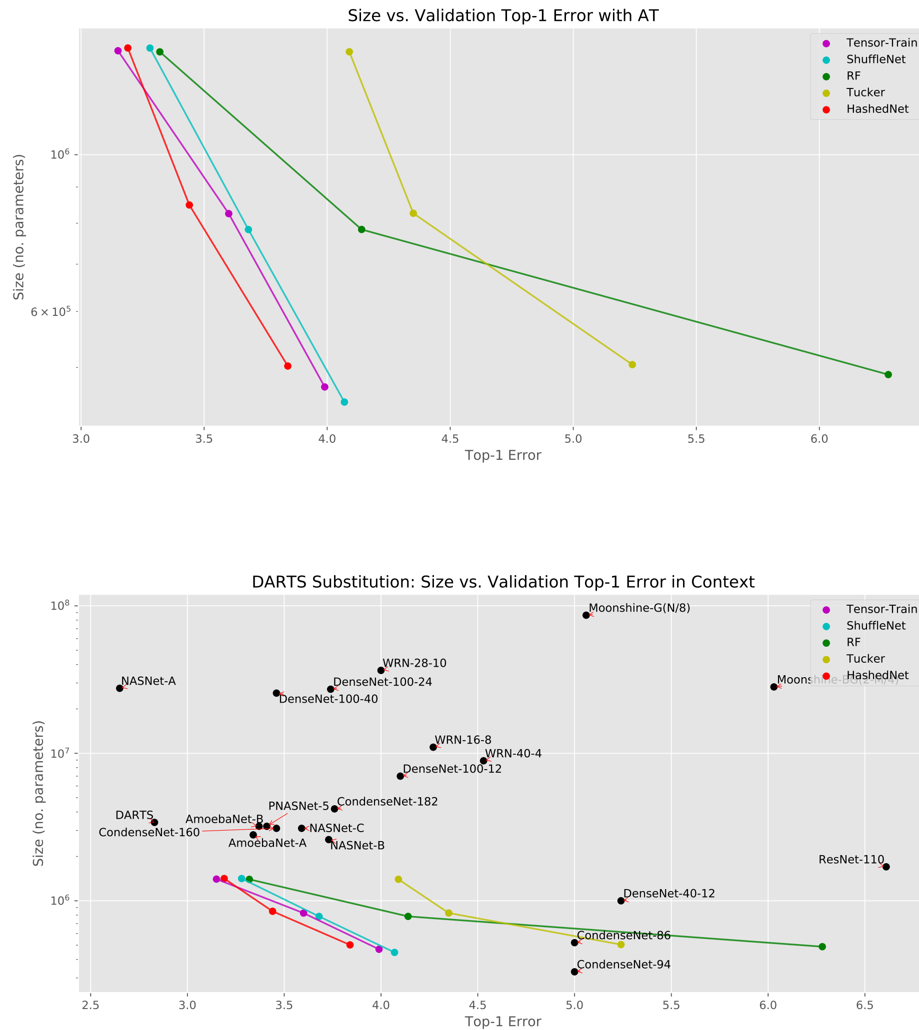


Figure 5.8: The relationship between top-1 error on the validation set and the number of parameters is plotted for experiments involving DARTS on CIFAR-10 trained with AT. Each substitute linear transform tested is indicated in the legend, minus ACDC as it failed to converge below 8% in any case. In the bottom figure we illustrate this performance in context with results from the literature on CIFAR-10. Networks compared against are recent networks presented in the literature as efficient, including: DenseNet (Huang et al., 2016a), GoogleNet (Szegedy et al., 2015), CondenseNet (Huang et al., 2017b), NASNet (Zoph et al., 2017), ResNet (He et al., 2016a), PNASNet (Liu et al., 2017), AmoebaNet (Real et al., 2017) and DARTS (Liu et al., 2018).

and ShuffleNet to compare on ImageNet. Also, despite its comparatively worse performance, we also ran experiments with the RF substitution as a baseline.

The results of the experiments are shown in Table 5.1 and in context with the literature in Figure 5.9. ImageNet is a more difficult problem, and we see that the performance is rapidly degraded as we reduce the number of parameters, although this appears to be the same trend observed with published networks in the literature (Howard et al., 2017).

We noted that the WRN-50-2 network we used in ImageNet experiments has a large linear layer to produce the logits used for classification. As the methods compared as substitutions here were all proposed originally as substitutions for linear layers, it seemed worth also compressing that layer. These are the experiments referred to in Table 5.1 and Figure 5.9 with the postfix "(+Linear)". They appear to follow the trend of compression in parameter reduction in Figure 5.9.

While the results on ImageNet do not exceed the Pareto frontier of the state of the art, the ability to take this established network and, just by substituting alternative convolutions, produce a wide array of networks at different levels of compression, with commensurate performance, is practically useful. Also, it indicates that thinking about the linear transforms we use in our convolutions could be worthwhile.

5.3.4 *Arithmetic Operations*

Mult-adds are a proxy for the run time of network that leave out relevant details, such as memory bandwidth or how sequential these operations might be, but they provide some indication of the resources required to run a network. As such, they are often quoted in the literature to compare networks. Here, we investigate the number of mult-adds used by the networks described in Section 5.3.1 with each of the convolutional substitutions we have proposed to investigate.

In Figure 5.10 the relationship between the number of parameters used by the different methods and the number of mult-adds is illustrated. As noted in Section 5.2.3, we approximate the number of mult-adds used by Tensor-Train and Tucker by the number used by the base network, despite it being likely that more efficient algorithms are possible. We cannot say much about these methods in terms of mult-add counts, they have been left out of Figures 5.11 and 5.12.

We chose the number of layers to be used by the ACDC substitution to fulfil a target parameter budget. The log-linear relationship between mult-adds and the number of parameters used by a single ACDC layer can be seen in Figure 5.10. However, with 12 layers, as used in the original paper (Moczulski et al., 2015), the number of mult-adds spent is only equal to the same-sized dense linear transform at 625 channels. In the

Table 5.1: Results training a WRN-50-2 on ImageNet with the proposed substitutions of the convolutions in the network with HashedNet, ShuffleNet, Tensor-Train and RF convolutions. Each method is tested at two compression ratios, designed to place each method in a close parameter budget. Compression is given as a percentage of the original model. Methods with "(+Linear)" appended have also replaced the final linear layer to produce the logits with a compressed linear transform. Methods from the literature are provided for comparison: WRN-50-2 (Zagoruyko and Komodakis, 2016), ShuffleNet (Zhang et al., 2017a), DenseNet (Huang et al., 2016a), GoogleNet (Szegedy et al., 2015), MobileNet (Howard et al., 2017), DecomposeMe (Alvarez and Salzmann, 2017), TRN (Kossaifi et al., 2017), ACDC (Moczulski et al., 2015) and TT (Novikov et al., 2015).

Model	Substitution	Parameters	Mult-Adds	Top-1 (%)	Compression (%) Size	Compression (%) Mult-Adds
WRN-50-2	ShuffleNet	6.04M	0.91G	29.73	8.77	8.00
WRN-50-2	ShuffleNet	17.72M	3.22G	26.93	25.72	28.22
WRN-50-2	ShuffleNet (+Linear)	4.38M	0.91G	30.61	6.35	7.98
WRN-50-2	RF	4.35M	0.53G	39.80	6.32	4.62
WRN-50-2	RF	17.55M	2.83G	25.44	25.48	24.77
WRN-50-2	HashedNet	4.35M	4.86G	33.45	6.32	42.59
WRN-50-2	HashedNet	17.61M	4.86G	24.52	25.56	42.59
WRN-50-2	HashedNet (+Linear)	2.47M	4.87G	35.08	3.58	42.67
WRN-50-2	Tensor-Train	4.34M	4.86G	33.24	6.30	42.59
WRN-50-2	Tensor-Train	17.58M	4.86G	24.89	25.52	42.59
WRN-50-2		68.9M	11G	21.9		
ShuffleNet		1.87M	0.14G	32.4		
ShuffleNet 2x		7.51M	0.53G	24.7		
DenseNet-201		20M	80G	22.6		
DenseNet-121		9M	6G	25.0		
GoogleNet		6.8M	1.55G	31.1		
MobileNet		4.2M	0.57G	29.4		
Dec_8^{512}			0.45G	33.2	46.5	53.8
VGG-19(TRN)		47.2M		31.2	34.13	
CaffeNet(ACDC)		9.7M		43.26	16.7	
VGG-16(TT)		18.65M		32.2	13.5	
VGG-19(TT)		24.0M		31.6	16.7	

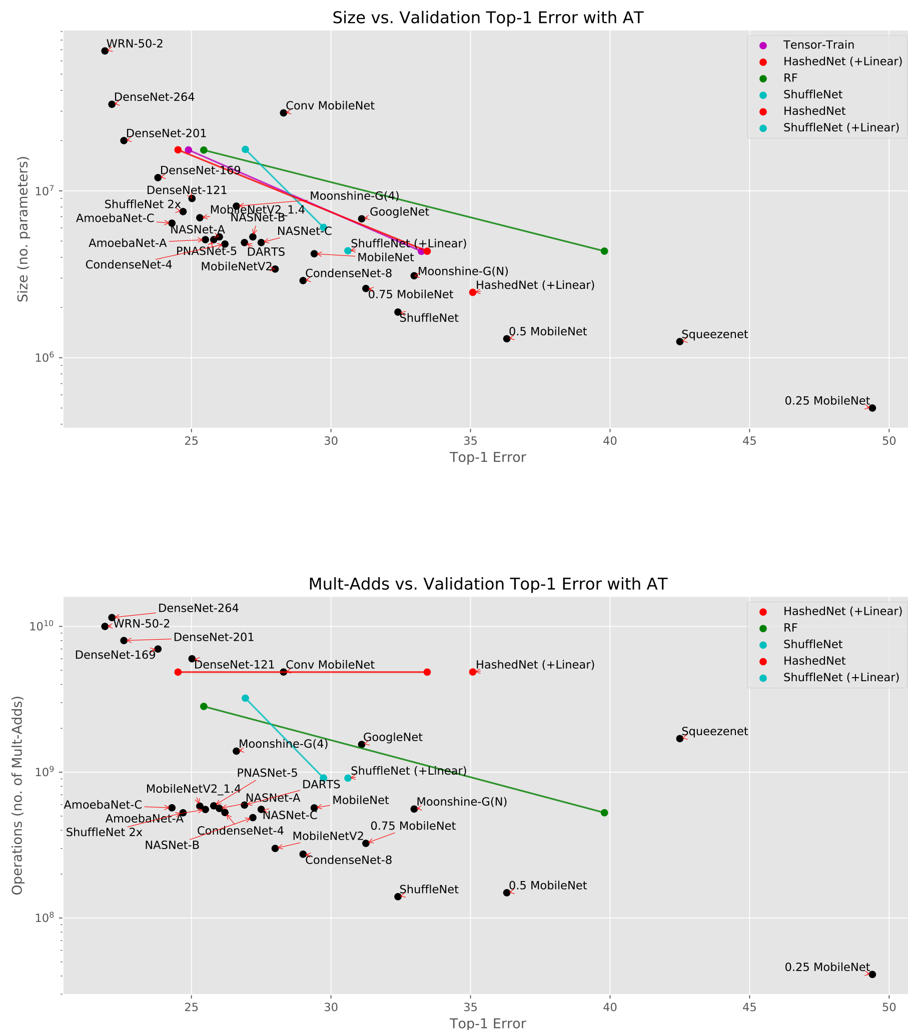


Figure 5.9: WRN-50-2 trained on *ImageNet* with AT and substituting HashedNet, ShuffleNet, Tensor-Train and RF linear transforms. Results are illustrate in context with results from the literature. While our results at different compression levels have a higher top-1 error than the state-of-the-art, the trend in top-1 error against parameter count matches the trend over all the networks illustrated. We also see that the RF and ShuffleNet substitutions allow us to explore a range of mult-add/top-1 error trade-offs in the lower figure. Networks compared against are recent networks presented in the literature as efficient, including: MobileNet (Howard et al., 2017), GoogleNet (Szegedy et al., 2015), SqueezeNet (Iandola et al., 2016), ShuffleNet (Zhang et al., 2017a), NASNet (Zoph et al., 2017), PNASNet (Liu et al., 2017), AmoebaNet (Real et al., 2017) and DARTS (Liu et al., 2018).

Figure 5.10: How the mult-adds used by different methods varies depending on the number of parameters they use for the WRN-28-10 network used in CIFAR-10 experiments. Many of the methods compared do not offer a lower computational cost versus the original separable convolution. In a WRN-28-10 the RF and ShuffleNet substitutions both offer a similar reduction in mult-adds. Unfortunately, an ACDC substitution uses significantly more mult-adds.

networks considered, only the very last layers have more than 625 channels, while in the original paper ACDC layers were *only* used on the final layers, of at least 1024 units. Due to this high mult-add cost, ACDC was left out of Figures 5.11 and 5.12.

Mult-add use is plotted against top-1 error on the validation set for WRN-28-10 using CIFAR-10 in Figure 5.11. Similar to the results from Section 5.3.3 with WRN-28-10 we see that the RF substitution is *just as efficient* as the more complex ShuffleNet block. However, in Figure 5.12 that same relationship is illustrated for DARTS on CIFAR-10, and we see significant benefits for the ShuffleNet block. It is able to reduce the mult-add count by 75% while being within 1% accuracy.

It is not possible to compare these results for mult-adds directly to other results in the literature as it is common to neglect reporting of mult-adds when reporting results on CIFAR-10. However, on ImageNet this comparison can be made, and the substitution experiments we have performed with WRN-50-2 are illustrated in Figure 5.9 and described in Table 5.1.

The RF block is again competitive. At the higher parameter budget, it outperforms the ShuffleNet substitution by 2%. However, at the lower budget the ShuffleNet substitution has 10% lower error, making the RF block seem impractical.

While the baseline RF block performs very well, it is worth noting the cases where it appears to be limited. In future experiments it would be worth investigating other substitutions with clear mult-add savings. For example, Moczulski et al. (2015) suggest a block-diagonal version of ACDC, which would be more similar to a ShuffleNet block.

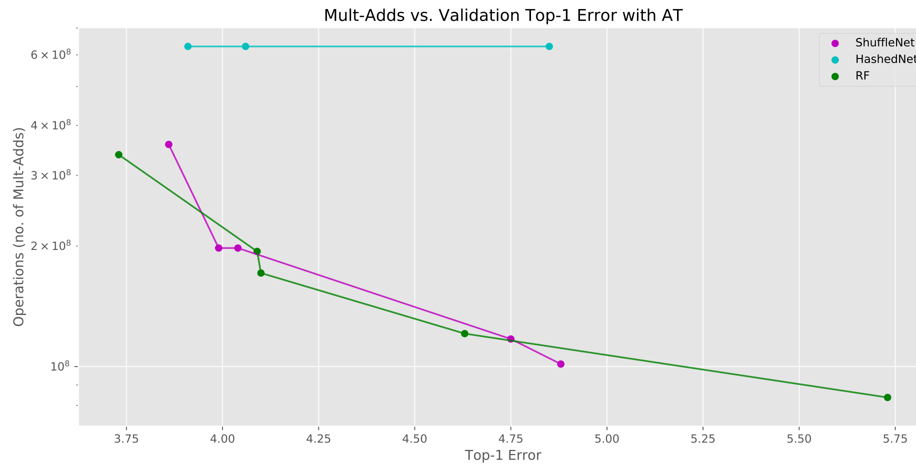


Figure 5.11: The relationship between top-1 error on the validation set and the number of mult-adds used by the network is plotted for experiments involving WRN-28-10 on CIFAR-10, trained with AT. Each substitute linear transform tested is indicated in the legend.

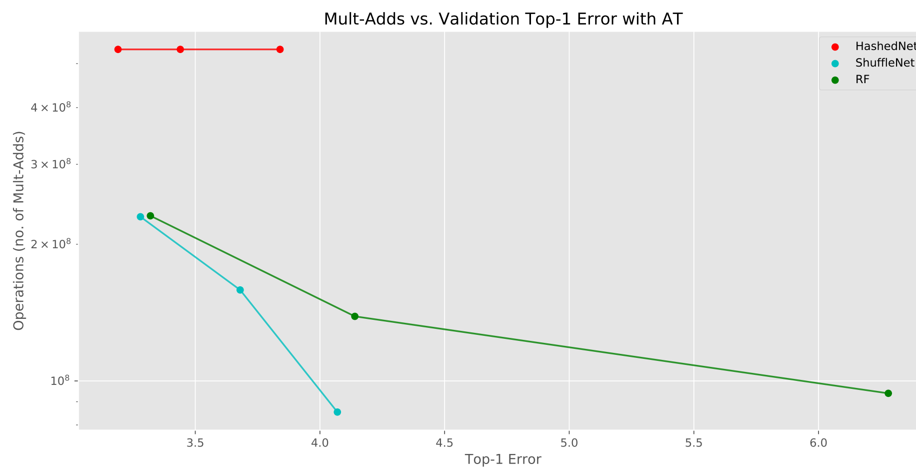


Figure 5.12: The relationship between top-1 error on the validation set and the number of mult-adds used by the network is plotted for experiments involving DARTS on CIFAR-10, trained with AT. Each substitute linear transform tested is indicated in the legend.

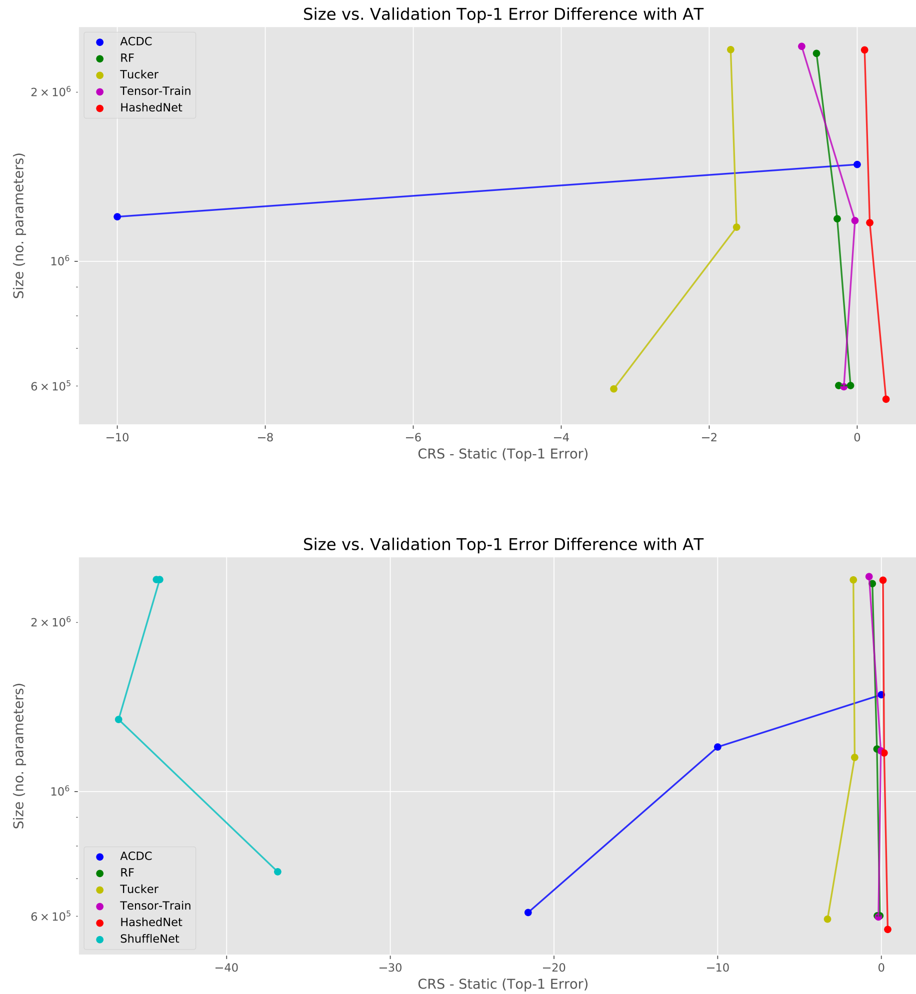


Figure 5.13: The difference in top-1 error on the validation set, when training with and without CRS weight decay, over all the substitution methods considered. For all methods apart from HashedNet, this form of weight decay scaling is beneficial; in other words, it results in a lower top-1 validation error.

5.3.5 *Compression Ratio Scaled Weight Decay*

To justify the use of CRS weight decay, we ran an ablation experiment, repeating the experiments on CIFAR-10 with WRN-28-10, but disabling CRS weight decay. In Figure 5.13 these results are illustrated. For almost all methods we see that there is a clear benefit. ShuffleNet simply fails to converge without it. However, for HashedNet we see that it is slightly detrimental.

Figure 5.14 illustrates the difference in top-1 validation error for HashedNet substitutions. We can see that the increase in top-1 error when using CRS weight decay is small, but consistent, with and without AT.

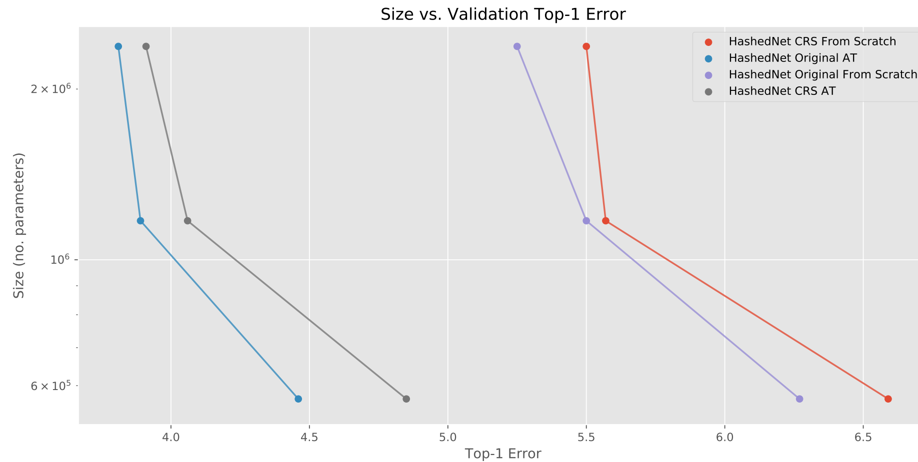


Figure 5.14: HashedNet top-1 validation error with and without AT, trained with and without CRS weight decay. We can see that, in all cases, top-1 validation error is slightly lower *without* CRS weight decay, for this substitution.

To investigate why this happens, Figure 5.15 illustrates the learning curves, top-1 error plots against the epoch during training, of these HashedNet substitute networks. The CRS weight decay stabilises training as we would hope, and the top-1 validation error is lower with it enabled *until the final stage* of the learning rate schedule. At this stage we can see the training top-1 error decreases faster when CRS weight decay is enabled. This overfitting is enough to cause a slight increase in top-1 validation error.

For ShuffleNet and ACDC substitutions, a decrease in the weight decay is necessary for convergence, and CRS weight decay scaling appears to work well. For other methods, such as Tensor-Train, Tucker or RF blocks, we see a consistent benefit. HashedNet appears to be the most stable of the methods considered, likely because it is a simple weight sharing scheme, so CRS weight decay causes a small amount of overfitting. Regardless, we leave it enabled for all experiments.

5.3.6 *Is Distillation Necessary?*

AT distillation is used in the experiments on CIFAR-10 using DARTS and in ImageNet experiments because we saw it universally improved top-1 validation error in the experiments with WRN-28-10 on CIFAR-10. Figure 5.16 illustrates the difference in top-1 error validation error in the WRN-28-10 experiments. Almost all methods benefit by more than 1%, which can be critical for a competitive top-1 error score on CIFAR-10, as seen in Figure 5.7.

Unlike methods, such as a hyperparameter search, that allow us to spend more computation time to achieve a better top-1 error, AT does not require tuning. We did

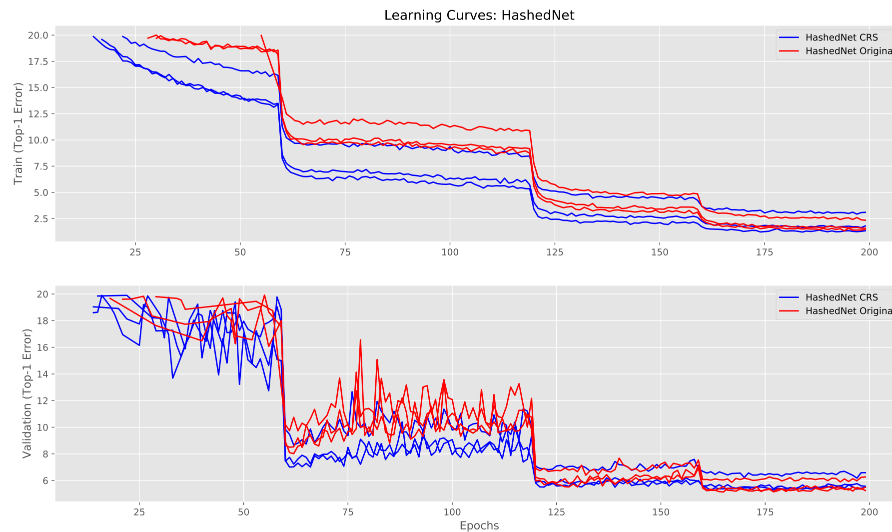


Figure 5.15: Learning curves for HashedNet substitution experiments, with and without CRS weight decay. When CRS weight decay is enabled the top-1 error is lower, on train and test, at every epoch until the final part of the learning rate schedule.

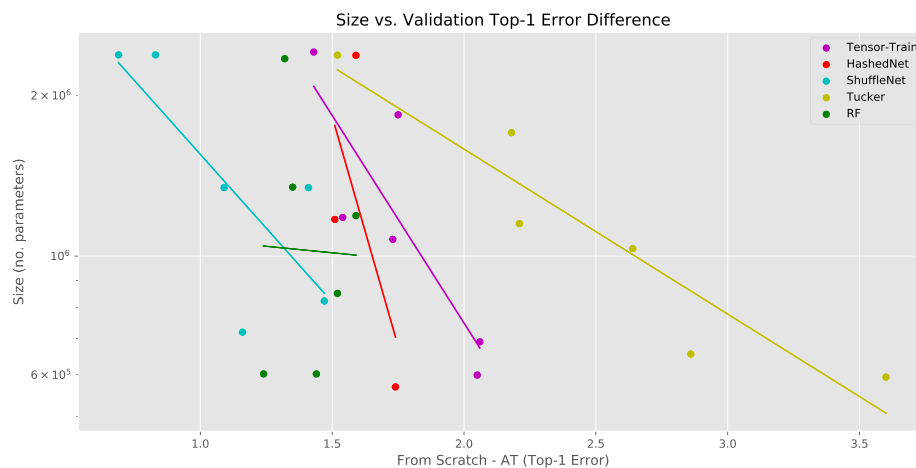


Figure 5.16: The difference in top-1 validation error with and without AT distillation is plotted for each substitute linear transform applied to WRN-28-10 on the CIFAR-10 classification problem. Over all the linear transform substitutions, AT lowers the top-1 validation error by a few percent. It has the greatest benefit in conjunction with the Tucker substitution, which has a higher overall top-1 validation error, as can be seen in Figure 5.7.

not tune any hyperparameters in any of the experiments presented here. For this reason, we can view AT as a way to compensate for an inadequate training routine.

We can see from the results presented in Figures 5.7 and 5.8 that the linear transform substitutions considered here are *capable* of learning the appropriate functions to implement a deep neural network. In our experiments, AT provides a way to demonstrate that potential. However, we acknowledge that this would incur a cost in a direct application of this work, but the cost is only that the networks presented here would take *at maximum* twice as long to train as the base network.

5.4 CONCLUSION

The use of dense randomly initialised matrices for linear transforms is universal in deep learning. While alternatives have been proposed in the literature, it was never clear whether these could be used in all possible layers. Previous experiments focused on replacing only the final fully connected layers, which are absent in recent architectures. This left a question unanswered: can deep neural networks function with linear transforms that may use fewer parameters and also possibly less arithmetic operations?

In this work we have demonstrated that neural networks can function with these substitutions in any of the convolutional layers making up modern deep learning architectures. This includes networks operating at the state of the art in image classification, the most lauded application of deep learning to date. These results were demonstrated on standard benchmark datasets, such as CIFAR-10 (Krizhevsky, 2009) and ImageNet (Deng et al., 2009), and compare favourably to results in the field.

We found that many competing linear transforms could achieve similar performance, despite striking differences in their parameterisation. A linear bottleneck only performs slightly worse in compression than the best linear transform we tested, which was HashedNet on CIFAR-10 and ShuffleNet on ImageNet. Particularly on ImageNet, at high compression, the ShuffleNet substitution has 2.84% lower top-1 error than all competing linear transform substitutions. This substitution offers a considerably lower number mult-adds in implementation compared with a HashedNet substitution.

These results are competitive with the state of the art, as shown in Table 5.1. This is surprising, because here we are only investigating the properties of alternative linear transforms. Our investigation operated under some constraints that it is possible to relax, to achieve greater compression in future work. For example, we use a uniformly scaled substitution in all layers, when pruning results show that all layers are likely not equally useful (Huang et al., 2017b). We are also constrained here to linear

reparameterizations, while deep learning is known to benefit from learning nonlinear functions.

We have also not fully explored the potential of the methods compared. Tensor-Train or Tucker decompositions are promising for larger efficiency gains, particularly on extremely high dimensional problems. We have shown that either may easily store, and learn by SGD, the functions necessary for deep learning, although the TT decomposition is easier to train.

Also, while the results of the ACDC substitution considered here were not as impressive as those of competing methods, such structured efficient linear layers could still bring large efficiency benefits. The permutations and diagonal matrices are similar to the composition of the linear ShuffleNet substitution. It seems likely that some modification, for example using block-diagonal rather than diagonal matrices (Moczulski et al., 2015) may be more effective. Alternatively, they could be massively faster on specialised computing hardware, such as the optical processors (Saade et al., 2015) suggested by Moczulski et al. (2015).

These results, in total, demonstrate ways in which deep learning may be performed more efficiently by orders of magnitude with existing hardware and software. As hardware and deep learning frameworks advance, as they already have, the efficiency of deep learning may advance quickly. These results may contribute to rapid design and experimentation of novel machine learning systems on low-power hardware, or enable new massive experiments with larger architectures.

In this chapter we have shown that the dense random matrices used in deep neural networks are incidental, and that many alternative linear transforms could fulfil the same purpose. We have demonstrated effective methods to train such transforms, and the relative efficiency benefits to using them. Each of the methods considered has also been compared for the consideration of future researchers, who may now make an informed decision about a basic building block in their deep neural network design.

6 CONCLUSION

The functionality of machine learning is often considered separately from the resources it consumes. Running the algorithms to perform inference with a given model consumes operations on a processor. Storing the model consumes space on a storage medium. Gathering data costs time and money on the part of the agent gathering that data. This thesis has investigated where such costs come from, how they affect the learning algorithms we use and how to produce more efficient algorithms.

The most resource intensive popular machine learning algorithms are deep neural networks. These are typically trained and operated on specialised hardware. For example, Zoph and Le (2016) reported using 22,400 GPU-hours to complete their architecture search experiments. A conservative estimate of the real energy cost of this training regime is 5.3MWh[1] at a cost of approximately $20,160[2]. This need only be spent once at training time, but performing inference still requires a similar GPU, putting it out of reach of mobile or embedded devices.

Machine learning need not be so resource intensive. Chapter 2 reviews relevant works, particularly that which demonstrate the massive redundancy of computation in deep neural networks. Substantial research has already been completed with the goal of making deep neural networks more efficient, and each success demonstrates some redundancy in the original concept. For example, pruning out weights or units demonstrates that networks are redundantly computing many activations that are not necessary. Networks that can operate with weights stored in only two binary values demonstrate that many of the bits used to store the 32-bit floating point weights are redundant.

6.1 SUMMARY

In Chapter 3 we have investigated how the practice of building a probabilistic model can incorporate resource constraints at test time. We present a protocol of model design in this setting, based on stochastic gradient methods and automatic differentiation, that is effective, flexible and scalable.

[1] Each K40 GPU used consumes 235W (Nvidia, 2015).
[2] The price per GPU-hour on AWS being $0.90 (Amazon, 2019).

While optimizing the parameters of our model, we can also learn a budget split over the features to be used. We have demonstrated that this may be applied in high dimensional problems, using a toy rotational MNIST example. Using the same method, we demonstrate resource efficiency by allocating resources in a real world sensor network example. Finally, by considering a budget over the bytes to spend storing an image, we were able to learn a quantisation matrix for JPEG compression automatically from data.

As machine learning algorithms become integrated with decision making systems, the models that we build must take into account constraints, such as budgets. The method presented here is a step toward such systems. Learning algorithms are part of the infrastructure of the internet, and making such infrastructure more efficient is an important goal.

The flexible probabilistic models we describe in Chapter 3 typically incorporate deep neural networks to learn nonlinear functions from data. Making these building blocks more efficient is therefore important to save resources in a large proportion of applied machine learning scenarios.

In Chapter 4 we presented a method to reduce the computational resources used by any deep convolutional neural network. By substituting convolutional blocks with cheaper alternatives and training the resulting network with recent distillation methods we were able to demonstrate state of the art compression. In addition, we found that this method worked without any tuning of hyperparameters or additional design work on the part of the machine learning engineer.

These results were replicated over a wide array of deep convolutional network sizes on CIFAR-10, CIFAR-100 (Krizhevsky, 2009) and ImageNet (Deng et al., 2009). We also found that we could reduce the computational and storage requirements of a state of the art network for semantic segmentation on the CityScapes dataset (Cordts et al., 2016).

These results demonstrated the value of substituting a nonlinear sequence of convolutions with a cheaper alternative, but what if we were to focus on the linear transforms making up each convolution? Matrix multiplication of an input vector by a matrix of floating point values is universal in deep learning, so a way to perform this function more efficiently would be a valuable advance to the field.

Various methods have been proposed in the literature for resource-efficient linear transforms. In Chapter 5 we proposed a novel comparison to apply these proposals in the convolutions of deep networks. Previous work had only demonstrated the use of such methods on large fully connected layers that are no longer common in modern networks. It was not clear if these methods would give any benefit, given the parameter sharing convolutional layers already apply.

Of the methods compared we found all were effective in reducing the stored size of the deep convolutional networks into which they were substituted. We were able to extend the Pareto boundary on CIFAR-10 (Krizhevsky, 2009) by substituting more efficient linear transforms into state of the art networks. In addition, the compression achieved was comparable to alternative compression methods in the literature. We found that this compression could be achieved with only an automatic scaling of the weight decay dependent on the compression ratio during training.

Several of the methods trained in this manner also provide substantially greater computational efficiency. The number of mult-adds used by trained networks was comparable to efficient networks promoted in the literature, even when the original network prior to substitution was not designed with efficiency in mind.

6.2 ADVANCES

In this thesis, we have demonstrated methods to improve the resource of efficiency of machine learning algorithms. In Chapter 3 we introduced a new method to deal with the cost of gathering test time data while learning a useful predictive algorithm, while in Chapters 4 and 5 we demonstrated methods to reduce the cost of the learning algorithms themselves. In all three Chapters we found that the resources consumed could be considerably reduced.

Resources consumed in the gathering of data do not typically take into account the learning algorithm. Building on previous work with linear models, we developed a simple, scalable framework for building this budgeting task into the machine learning system. In Chapter 3 we demonstrated the effectiveness of this framework on a sensor network problem and by automatically learning the quantisation matrices required for the JPEG algorithm from single images; improving on the hardcoded quantisation matrices typically used. These budget considerations are a novel problem, and we demonstrate a flexible solution to that problem.

Compression of complex nonlinear models such as deep neural networks can be difficult and require extra engineering, such as using a pruning algorithm (Han et al., 2015). In Chapter 4 we demonstrate state of the art compression that operates using a rote substitution on an existing network; requiring a minimum of engineering effort. This provides a simple compression method that is applicable without esoteric hardware or software. Prior to this work, many experiments may have been required to tune an efficient architecture. Using the method described in Chapter 4, one may be all that is required.

Dense randomly initialised matrices are a major building block of deep neural networks. In Chapter 5 we demonstrate that many alternatives to such matrices are

readily applicable in modern convolutional architectures. Given the fundamental nature of such matrices, this advance demonstrates a direction to improve the efficiency of all deep neural networks, either at training or test time. In addition, using specialised hardware (Saade et al., 2015), allows for extremely efficient machine learning that could enable novel research in all machine learning applications. At time of writing, this is the only work that has demonstrated the transformational potential of replacing dense randomly initialised matrices. When considered in combination with the continued improvements in computing capabilities, existing experiments will be cheaper and radical new experiments will become possible.

6.3 FUTURE WORK

The results of Chapters 4 and 5 further reinforce the known redundancy of neural networks. At the same time, both investigate portable ideas to reduce said redundancy, which are relevant in many settings where deep neural networks are applied.

Greater efficiency benefits are likely possible. In our comparison experiments we were constrained to use a substitution at each layer with the same compression settings, but results in pruning, such as those detailed in Section 2.3, demonstrate that we could do better by setting the compression rates dynamically. Or, if we did not constrain ourselves to linear substitutions we may be able to achieve greater representational capacity at a lower cost, in the manner of the experiments of Chapter 4.

Deep neural networks can therefore be implemented and trained much more efficiently, with relatively small changes to existing practice. As neural networks become less redundant, and computational resources become cheaper, we should expect to see a proliferation of learning algorithms. The work on efficiency we present here provides a basis for such efforts.

In Chapter 5 we demonstrated that tensor decompositions provide efficient representations for storage of convolutional tensors, even in state of the art networks. This implies that such networks could be implemented entirely using tensor decompositions, such as Tensor-Train (Garipov et al., 2016) or the transformations presented in Tensor Regression Networks (Kossaifi et al., 2017). In either cases, a linear transform can be applied *while maintaining a decomposed representation*. The potential storage and efficiency benefits could be far greater than we present here, or it could allow networks to operate on much higher dimensional data than that over which deep neural networks are typically able to operate.

The storage or efficiency benefits presented in Chapter 4 may also be substantially improved by incorporating additional compression methods. We store the weights at 32-bit precision, while far fewer bits may be necessary, and it is likely we would see

substantial benefits by applying a pruning algorithm. As the method we present is agnostic to the network it is applied on, it works on any deep convolutional network, we can imagine that it will remain a useful abstraction in efficient deep convolutional network design.

We might also expect that considering resource constraints at test time to be a concern to designers of probabilistic models in future. Systems with the functionality to learn over a smaller dataset at test time, often referred to as *meta-learning systems*, have become more popular in recent years (Antoniou et al., 2019). The construction of such models under a budget constraint is precisely the problem that we consider in Chapter 3 and our investigation is likely to be of use in future work.

6.4 FINAL COMMENTS

Deep neural networks are a flexible parametric model, providing a method to learn complex functional relationships on large datasets. The functional relationship between an image and its associated class label being a famous example. No other algorithm is competitive in learning such a relationship and then being able to perform that task in a fraction of a second at test time.

However, in doing so, these algorithms carry out massive redundant computations. We would like to take advantage of these flexible, scalable algorithms using a minimum of resources.

In this thesis we have demonstrated simple modifications to the training protocols for deep neural networks that improve efficiency. We have also considered how a budget may be integrated into probabilistic models that incorporate deep neural networks to learn functional relationships. This will enable applications on larger datasets, on smaller hardware and using fewer resources.

LIST OF FIGURES

Figure 2.1 “Reduction of convolutional layer evaluation to matrix multiplication. Our idea is to leave only a subset of rows (defined by a perforation mask) in the data matrix M and to interpolate the missing output values.”(Figurnov et al., 2016) 13

Figure 2.2 More recent architectures are deeper and have skip connections, as in b, but they also rely on repeated 3x3 and pooling, in the style of Springenberg et al. (2014). Further, fully connected final layers like those in a are now replaced by average pooling as the final operation. This works thanks to strided convolutions reducing the size of the activation map at stages throughout the network. 21

Figure 2.3 Grouped convolutions apply independent filters over an input grouped by channel. 22

Figure 2.4 “Illustration of the first four layers of an MSDNet with three scales. The horizontal direction corresponds to the layer direction (depth) of the network. The vertical direction corresponds to the scale of the feature maps. Horizontal arrows indicate a regular convolution operation, whereas diagonal and vertical arrows indicate a strided convolution operation.” (Huang et al., 2017a) 24

Figure 2.5 “The evolution of the layers with the training epochs in the information plane, for different training samples. On the left - 5% of the data, middle - 45% of the data, and right - 85% of the data. The colors indicate the number of training epochs with Stochastic Gradient Descent from 0 to 10000. The network architecture was fully connected layers, with widths: input=12-10-8-6-4-2-1=output.” (Shwartz-Ziv and Tishby, 2017) 31

Figure 3.1 On the left the training set $\{\mathbf{X}^n\}_{n=1}^{N_{\text{train}}}$ is shown, illustrating the separation of features within each datapoint into contexts. These can be indexed by c: $\mathbf{x}_c^n{}_{n=1}^{N_{\text{train}}}$. In this learning process, a budget B is considered during training. Training then produces not just the trained *model*, but also a *budget split* to use at test time. The budget split then affects the noise incident on our trained model at test test time through the application of a real budget. The final result of this process are the predictions to be used in an application. 36

Figure 3.2 A supervised task with noisy inputs $\tilde{\mathbf{X}}^n$, noise-free true inputs $\mathbf{X}^n$ and targets $\mathbf{y}^n$. At training time we observe $\mathbf{X}^n$ and $\mathbf{y}^n$, while at test time we have only $\tilde{\mathbf{X}}^n$. 41

Figure 3.3 A stochastic computation graph (Schulman et al., 2015) describing the inference algorithm applied to the model described in Figure 3.2. Inputs x_{ij}^n at training time are optionally processed by a statistic network, parameterised by $\boldsymbol{\phi}$, to create a representation h_{ij}^n. Taking the mean of this representation produces a statistic used for classification, which passes through a noising function, as described in Equation 3.7, to produce $\tilde{\mathbf{x}}_j^n$. This is then used to produce a prediction $\tilde{\mathbf{y}}^n$ which, along with a target $\mathbf{y}^n$, produces a scalar loss L^n. As every step in this graph is differentiable, we are able to then optimise θ, $\boldsymbol{\phi}$ and $\boldsymbol{\gamma}$ with respect to this loss. . 43

Figure 3.4 An example statistic network, gathering observations x_1, x_2 and x_3 to make a statistic in the representation learnt by the network (Edwards and Storkey, 2017). 44

Figure 3.5 Rotational MNIST input (left) and output (right). Each context is a digit class, and at test time the budget decides how many images from each class we receive. 46

Figure 3.6 Left: comparing over different sparsity settings and a total budget of 10, the distribution of total budget required when uniformly split to match the performance of a gradient optimised allocation. Right: histogram of Bayesian optimised budget splits with vertical lines for the competing methods considered. 47

Figure 3.7 Illustration of the Tropical Atmospheric Ocean (TAO) array sensors for monitoring the El Niño event (Lichman, 2013). . . . 48

Figure 3.8 Normalised sensor placement of the subset of sensors used in the experiment on the TAO dataset described in Section 3.3.2. . . 49
Figure 3.9 The extra cost of using a uniform budget allocation over one found by gradient optimisation. Each imputation task is the problem of inferring the observations at any buoy. Buoys 2 and 8 are from proximal sensors, and learn budgets preferring data from each other. 49
Figure 3.10 Illustrating the budget splits learnt when imputing the observations at buoy 2 and buoy 8 on the TAO sensor network imputation experiment investigated in Section 3.3.2. In both cases, the most proximal buoy, as shown in Figure 3.8 is favoured over all others. 50
Figure 3.11 The 2D discrete cosine transform decomposes an 8 by 8 patch using a linear combination of these patterns (Wikimedia, 2019). These are referred to as the 2D DCT basis functions. 51
Figure 3.12 Learnt quantisation matrices are compared to the quantisation matrices prescribed by JPEG; "Probability of Preference" over a JPEG encoding of the same image. The PieApp pretrained perceptual error network (Prashnani et al., 2018) is used to give a probability of preference of the learnt matrix. Linear regression was used to estimate the perceptual preference if the file size were equal and these values are plotted against the prescribed budget. 55
Figure 3.13 Comparing the perceptual preference of learnt quantisation matrices when they are learnt either by optimising perceptual error directly, or implicitly using an MLP. Different points correspond to different images. 56
Figure 3.14 Example images created using quantisation matrices that are stored at comparable file sizes (prescribing an exact file size is not possible due to unpredictable lossless encoding). 57
Figure 4.1 "Sum of absolute values attention maps F_{sum} over different levels of a network trained for face recognition. Mid-level attention maps have higher activation level around eyes, nose and lips, high-level activations correspond to the whole face." (Zagoruyko and Komodakis, 2017) 66
Figure 4.2 Example of a convolution on a 2D input array, passing a 2×2 filter over the input space and returning 6 output values (Goodfellow et al., 2016, p.325). 67

Figure 4.3 In (a), a grouped convolution operates by passing independent filters over the tensor after it is separated into g groups over the channel dimension; as each of the g filters needs only to operate over N/g channels, this reduces the parameter cost of the layer by a factor of g. These can be composed into the blocks illustrated in (b). The *Grouped + Pointwise* block ($G(g)$) substitutes a $k \times k$ convolution with a grouped convolution followed by a pointwise (1×1) convolution, repeating this twice. To reduce parameters further, a pointwise *Bottleneck* can be used before the Grouped + Pointwise convolution ($BG(b, g)$). 70

Figure 4.4 Test Error vs. (a) Number of parameters and (b) mult-adds for student networks learnt with attention transfer on CIFAR-10. Note that the x-axes are log-scaled. Points on the red curve correspond to networks with S convolutional blocks and reduced architectures. All other networks have the same WRN-40-2 architecture as the teacher but with cheap convolutional blocks: G (green), B (blue), and BG (cyan). The blocks are described in Table 4.1. Notice that the student networks with cheap blocks outperform those with smaller architectures and standard convolutions for a given parameter budget or mult-add budget. . . 74

Figure 4.5 The progression of the different components of the loss during training, when using attention transfer. "Blocks" refer to the blockwise structure of a Wide ResNet (Zagoruyko and Komodakis, 2016), higher indices proceeding deeper into the network, the student network used a grouped+bottleneck block with the same architecture of depth 40 and width 2. Each AT component has been scaled by the appropriate β value used during training. 76

Figure 4.6 Progress of components of the loss function during training, as in Figure 4.5, but the student used here has precisely the same structure as the teacher. 76

Figure 4.7 Comparing top-1 error (%) to parameter count (top) and multiply-add operations (bottom). The two ResNet-34 networks presented in Table 4.4 are tagged “Moonshine”. Networks compared against are recent networks presented in the literature as efficient, including: MobileNet (Howard et al., 2017), GoogleNet (Szegedy et al., 2015), SqueezeNet (Iandola et al., 2016), ShuffleNet (Zhang et al., 2017a), NASNet (Zoph et al., 2017), PNASNet (Liu et al., 2017), AmoebaNet (Real et al., 2017) and DARTS (Liu et al., 2018). 79

Figure 5.1 "An illustration of a neural network with random weight sharing under compression factor $\frac{1}{4}$. The $16+9=24$ virtual weights are compressed into 6 real weights. The colors represent matrix elements that share the same weight value." (Chen et al., 2015) 92

Figure 5.2 The effect on percentage of weights excluded depending on compression ration c, tested for different values of N_v, the number of elements in the virtual weight matrix, indicated in the legend. At the compression levels we are interested in, 20% of the original number of weights, we can see that the number of weights excluded is low. 93

Figure 5.3 "ShuffleNet Units. a) bottleneck unit (He et al., 2016a) with depthwise convolution (DWConv) (Chollet, 2016; Howard et al., 2017); b) ShuffleNet unit with pointwise group convolution (GConv) and channel shuffle; c) ShuffleNet unit with stride = 2." (Zhang et al., 2017a) 96

Figure 5.4 The first GConv in Figure 5.3 is illustrated in *A*. The proceeding channel shuffle connects these independent groups in *B*. Passing through the final GConv block produces the approximately dense random matrix in *C*. 96

Figure 5.5 The parameter cost of a WRN-28-10 after substitution by the methods listed in the legend, varying the tunable parameter of each over a normalised range. We design experiments over a parameter count range such that all methods illustrated will have support, which here is limited by the maximum size of the Linear ShuffleNet and the minimum size of the RF substitution. 98

Figure 5.6 Results of a grid search over settings when using Tensor-Train and Tucker decomposition substitutions in a ResNet-18-like architecture optimised for fast experiments, as described in Section 5.3.1. Figures c and a show the relative top-1 error achieved over the gride search settings. Figures d and b show the variation in compression ratio over the same range. We can see that, while the compression ratio is higher as the dimensionality of the tensor increases, the top-1 error possible begins to suffer. 103

Figure 5.7 The relationship between top-1 error on the validation set and the number of parameters is plotted for experiments involving WRN-28-10 on CIFAR-10, for experiments using AT and without. Each substitute linear transform tested is indicated in the legend. On this problem, both Tensor-Train and HashedNet substitutions are able to achieve the highest rates of compression. At lower compression settings, all methods compared achieve comparable top-1 error. Note that ACDC was unstable for larger networks and so we only plot results for the smallest parameter budget. 105

Figure 5.8 The relationship between top-1 error on the validation set and the number of parameters is plotted for experiments involving DARTS on CIFAR-10 trained with AT. Each substitute linear transform tested is indicated in the legend, minus ACDC as it failed to converge below 8% in any case. In the bottom figure we illustrate this performance in context with results from the literature on CIFAR-10. Networks compared against are recent networks presented in the literature as efficient, including: DenseNet (Huang et al., 2016a), GoogleNet (Szegedy et al., 2015), CondenseNet (Huang et al., 2017b), NASNet (Zoph et al., 2017), ResNet (He et al., 2016a), PNASNet (Liu et al., 2017), AmoebaNet (Real et al., 2017) and DARTS (Liu et al., 2018). . . 106

Figure 5.9 WRN-50-2 trained on *ImageNet* with AT and substituting HashedNet, ShuffleNet, Tensor-Train and RF linear transforms. Results are illustrate in context with results from the literature. While our results at different compression levels have a higher top-1 error than the state-of-the-art, the trend in top-1 error against parameter count matches the trend over all the networks illustrated. We also see that the RF and ShuffleNet substitutions allow us to explore a range of mult-add/top-1 error trade-offs in the lower figure. Networks compared against are recent networks presented in the literature as efficient, including: MobileNet (Howard et al., 2017), GoogleNet (Szegedy et al., 2015), SqueezeNet (Iandola et al., 2016), ShuffleNet (Zhang et al., 2017a), NASNet (Zoph et al., 2017), PNASNet (Liu et al., 2017), AmoebaNet (Real et al., 2017) and DARTS (Liu et al., 2018). 109

Figure 5.10 How the mult-adds used by different methods varies depending on the number of parameters they use for the WRN-28-10 network used in CIFAR-10 experiments. Many of the methods compared do not offer a lower computational cost versus the original separable convolution. In a WRN-28-10 the RF and ShuffleNet substitutions both offer a similar reduction in mult-adds. Unfortunately, an ACDC substitution uses significantly more mult-adds. 110

Figure 5.11 The relationship between top-1 error on the validation set and the number of mult-adds used by the network is plotted for experiments involving WRN-28-10 on CIFAR-10, trained with AT. Each substitute linear transform tested is indicated in the legend. 111

Figure 5.12 The relationship between top-1 error on the validation set and the number of mult-adds used by the network is plotted for experiments involving DARTS on CIFAR-10, trained with AT. Each substitute linear transform tested is indicated in the legend.111

Figure 5.13 The difference in top-1 error on the validation set, when training with and without CRS weight decay, over all the substitution methods considered. For all methods apart from HashedNet, this form of weight decay scaling is beneficial; in other words, it results in a lower top-1 validation error. 112

Figure 5.14 HashedNet top-1 validation error with and without AT, trained with and without CRS weight decay. We can see that, in all cases, top-1 validation error is slightly lower *without* CRS weight decay, for this substitution. 113

Figure 5.15 Learning curves for HashedNet substitution experiments, with and without CRS weight decay. When CRS weight decay is enabled the top-1 error is lower, on train and test, at every epoch until the final part of the learning rate schedule. 114

Figure 5.16 The difference in top-1 validation error with and without AT distillation is plotted for each substitute linear transform applied to WRN-28-10 on the CIFAR-10 classification problem. Over all the linear transform substitutions, AT lowers the top-1 validation error by a few percent. It has the greatest benefit in conjunction with the Tucker substitution, which has a higher overall top-1 validation error, as can be seen in Figure 5.7. . . . 114

BIBLIOGRAPHY

Abadi, M., Agarwal, A., Barham, P., Brevdo, E., Chen, Z., Citro, C., Corrado, G. S., Davis, A., Dean, J., Devin, M., Ghemawat, S., Goodfellow, I., Harp, A., Irving, G., Isard, M., Jia, Y., Jozefowicz, R., Kaiser, L., Kudlur, M., Levenberg, J., Mané, D., Monga, R., Moore, S., Murray, D., Olah, C., Schuster, M., Shlens, J., Steiner, B., Sutskever, I., Talwar, K., Tucker, P., Vanhoucke, V., Vasudevan, V., Viégas, F., Vinyals, O., Warden, P., Wattenberg, M., Wicke, M., Yu, Y., and Zheng, X. (2015). TensorFlow: Large-scale machine learning on heterogeneous systems. Software available from tensorflow.org. (Cited on page 23.)

Ahmed, N., Natarajan, T., and Rao, K. R. (1974). Discrete cosine transform. *IEEE transactions on Computers*, 100(1):90–93. (Cited on page 52.)

Al-Rfou, R., Alain, G., Almahairi, A., Angermueller, C., Bahdanau, D., Ballas, N., Bastien, F., Bayer, J., Belikov, A., Belopolsky, A., Bengio, Y., Bergeron, A., Bergstra, J., Bisson, V., Bleecher Snyder, J., Bouchard, N., Boulanger-Lewandowski, N., Bouthillier, X., de Brébisson, A., Breuleux, O., Carrier, P.-L., Cho, K., Chorowski, J., Christiano, P., Cooijmans, T., Côté, M.-A., Côté, M., Courville, A., Dauphin, Y. N., Delalleau, O., Demouth, J., Desjardins, G., Dieleman, S., Dinh, L., Ducoffe, M., Dumoulin, V., Ebrahimi Kahou, S., Erhan, D., Fan, Z., Firat, O., Germain, M., Glorot, X., Goodfellow, I., Graham, M., Gulcehre, C., Hamel, P., Harlouchet, I., Heng, J.-P., Hidasi, B., Honari, S., Jain, A., Jean, S., Jia, K., Korobov, M., Kulkarni, V., Lamb, A., Lamblin, P., Larsen, E., Laurent, C., Lee, S., Lefrancois, S., Lemieux, S., Léonard, N., Lin, Z., Livezey, J. A., Lorenz, C., Lowin, J., Ma, Q., Manzagol, P.-A., Mastropietro, O., McGibbon, R. T., Memisevic, R., van Merriënboer, B., Michalski, V., Mirza, M., Orlandi, A., Pal, C., Pascanu, R., Pezeshki, M., Raffel, C., Renshaw, D., Rocklin, M., Romero, A., Roth, M., Sadowski, P., Salvatier, J., Savard, F., Schlüter, J., Schulman, J., Schwartz, G., Serban, I. V., Serdyuk, D., Shabanian, S., Simon, E., Spieckermann, S., Subramanyam, S. R., Sygnowski, J., Tanguay, J., van Tulder, G., Turian, J., Urban, S., Vincent, P., Visin, F., de Vries, H., Warde-Farley, D., Webb, D. J., Willson, M., Xu, K., Xue, L., Yao, L., Zhang, S., and Zhang, Y. (2016). Theano: A Python framework for fast computation of mathematical expressions. *arXiv e-prints*, abs/1605.02688. (Cited on pages 23 and 45.)

Albericio, J., Judd, P., Hetherington, T., Aamodt, T., Jerger, N. E., and Moshovos, A. (2016). Cnvlutin: Ineffectual-neuron-free deep neural network computing. In *Computer Architecture (ISCA), 2016 ACM/IEEE 43rd Annual International Symposium on*, pages 1–13. IEEE. (Cited on page 13.)

Almahairi, A., Ballas, N., Cooijmans, T., Zheng, Y., Larochelle, H., and Courville, A. C. (2015). Dynamic capacity networks. *CoRR*, abs/1511.07838. (Cited on page 23.)

Alvarez, J. M. and Petersson, L. (2016). DecomposeMe: Simplifying convnets for end-to-end learning. *CoRR*, abs/1606.05426. (Cited on page 86.)

Alvarez, J. M. and Salzmann, M. (2016). Learning the number of neurons in deep networks. *CoRR*, abs/1611.06321. (Cited on pages 7, 10, and 32.)

Alvarez, J. M. and Salzmann, M. (2017). Compression-aware Training of Deep Networks. *ArXiv e-prints*. (Cited on pages 26, 32, 86, and 108.)

Amazon (2019). Amazon EC2 – P2 instances. https://aws.amazon.com/ec2/instance-types/p2/. Accessed: 3rd March 2019. (Cited on page 117.)

Antoniou, A., Edwards, H., and Storkey, A. (2019). How to train your MAML. In *International Conference on Learning Representations*. (Cited on page 121.)

Ba, J., Hinton, G., Mnih, V., Leibo, J. Z., and Ionescu, C. (2016). Using Fast Weights to Attend to the Recent Past. *ArXiv e-prints*. (Cited on page 25.)

Ba, L. J. and Caruana, R. (2014). Do Deep Nets Really Need to be Deep? In *Advances in Neural Information Processing Systems*. (Cited on pages 28, 61, 63, 64, and 66.)

Ballester, R. (2019). tntorch: Tensor network learning with PyTorch. https://github.com/rballester/tntorch. Accessed: 14th February 2019. (Cited on page 99.)

Bengio, Y., Léonard, N., and Courville, A. C. (2013). Estimating or propagating gradients through stochastic neurons for conditional computation. *CoRR*, abs/1308.3432. (Cited on page 15.)

Blundell, C., Cornebise, J., Kavukcuoglu, K., and Wierstra, D. (2015). Weight Uncertainty in Neural Networks. *ArXiv e-prints*. (Cited on page 11.)

Bojarski, M., Choromanska, A., Choromanski, K., Fagan, F., Gouy-Pailler, C., Morvan, A., Sakr, N., Sarlós, T., and Atif, J. (2016). Structured adaptive and random spinners for fast machine learning computations. *CoRR*, abs/1610.06209. (Cited on pages 1, 27, and 42.)

Bonnet, G. (1964). Transformations des signaux aléatoires a travers les systemes non linéaires sans mémoire. *Annals of Telecommunications*, 19(9):203–220. (Cited on page 37.)

Brochu, E., Cora, V. M., and de Freitas, N. (2010). A tutorial on bayesian optimization of expensive cost functions, with application to active user modeling and hierarchical reinforcement learning. *CoRR*, abs/1012.2599. (Cited on page 46.)

Brock, A., Lim, T., Ritchie, J. M., and Weston, N. (2017). SMASH: one-shot model architecture search through hypernetworks. *CoRR*, abs/1708.05344. (Cited on page 25.)

Chai, S., Raghavan, A., Zhang, D. C., Amer, M. R., and Shields, T. J. (2017). Low precision neural networks using subband decomposition. *CoRR*, abs/1703.08595. (Cited on page 17.)

Chen, T., Goodfellow, I. J., and Shlens, J. (2015). Net2Net: Accelerating learning via knowledge transfer. *CoRR*, abs/1511.05641. (Cited on page 28.)

Chen, W., Wilson, J., Tyree, S., Weinberger, K. Q., and Chen, Y. (2016). Compressing convolutional neural networks in the frequency domain. In *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 1475–1484. ACM. (Cited on page 84.)

Chen, W., Wilson, J. T., Tyree, S., Weinberger, K. Q., and Chen, Y. (2015). Compressing Neural Networks with the Hashing Trick. *ArXiv e-prints*. (Cited on pages 84, 87, 91, 92, and 126.)

Chetlur, S., Woolley, C., Vandermersch, P., Cohen, J., Tran, J., Catanzaro, B., and Shelhamer, E. (2014). cuDNN: Efficient primitives for deep learning. *CoRR*,

abs/1410.0759. (Cited on pages 20 and 26.)

Choi, Y., El-Khamy, M., and Lee, J. (2016). Towards the limit of network quantization. *CoRR*, abs/1612.01543. (Cited on page 9.)

Chollet, F. (2016). Xception: Deep learning with depthwise separable convolutions. *CoRR*, abs/1610.02357. (Cited on pages 22, 64, 73, 88, 96, and 126.)

Collins, M. D. and Kohli, P. (2014). Memory bounded deep convolutional networks. *CoRR*, abs/1412.1442. (Cited on pages 7 and 10.)

Contardo, G., Denoyer, L., and Artières, T. (2016). Recurrent neural networks for adaptive feature acquisition. In Hirose, A., Ozawa, S., Doya, K., Ikeda, K., Lee, M., and Liu, D., editors, *Neural Information Processing*, pages 591–599, Cham. Springer International Publishing. (Cited on pages 1 and 42.)

Coppersmith, D. and Winograd, S. (1990). Matrix multiplication via arithmetic progressions. *Journal of Symbolic Computation*, 9(3):251 – 280. (Cited on page 89.)

Cordts, M., Omran, M., Ramos, S., Rehfeld, T., Enzweiler, M., Benenson, R., Franke, U., Roth, S., and Schiele, B. (2016). The cityscapes dataset for semantic urban scene understanding. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*. (Cited on pages 62, 80, and 118.)

Courbariaux, M. and Bengio, Y. (2016). Binarized neural networks: Training deep neural networks with weights and activations constrained to +1 or -1. *CoRR*, abs/1602.02830. (Cited on pages 15 and 16.)

Courbariaux, M., Bengio, Y., and David, J. (2015). Binaryconnect: Training deep neural networks with binary weights during propagations. *CoRR*, abs/1511.00363. (Cited on pages 15, 17, and 30.)

Cover, T. and Thomas, J. (2006). *Elements of Information Theory*. Wiley. (Cited on page 65.)

Crowley, E. J., Gray, G., and Storkey, A. (2017). Moonshine: Distilling with Cheap Convolutions. *ArXiv e-prints*. (Cited on pages 28, 33, 61, and 63.)

Crowley, E. J., Turner, J., Storkey, A., and O'Boyle, M. (2018). A Closer Look at Structured Pruning for Neural Network Compression. *arXiv e-prints*, page arXiv:1810.04622. (Cited on pages 12 and 32.)

Deng, J., Dong, W., Socher, R., Li, L.-J., Li, K., and Fei-Fei, L. (2009). ImageNet: A Large-Scale Hierarchical Image Database. In *CVPR09*. (Cited on pages 8, 25, 32, 62, 67, 77, 85, 99, 100, 115, and 118.)

Denil, M., Shakibi, B., Dinh, L., Ranzato, M., and de Freitas, N. (2013). Predicting Parameters in Deep Learning. In *Advances in Neural Information Processing Systems*. (Cited on pages 6, 30, and 63.)

Denton, E. L., Zaremba, W., Bruna, J., LeCun, Y., and Fergus, R. (2014). Exploiting linear structure within convolutional networks for efficient evaluation. In *Advances in Neural Information Processing Systems*, pages 1269–1277. (Cited on pages 25, 26, and 86.)

Devries, T. and Taylor, G. W. (2017). Improved regularization of convolutional neural networks with cutout. *CoRR*, abs/1708.04552. (Cited on pages 100 and 101.)

Devroye, L. (1986). *Non-Uniform Random Variate Generation*. Springer-Verlag, New York, 1st edition. (Cited on page 53.)

Dieleman, S., Schlüter, J., Raffel, C., Olson, E., Sønderby, S. K., Nouri, D., Maturana, D., Thoma, M., Battenberg, E., Kelly, J., Fauw, J. D., Heilman, M., de Almeida, D. M., McFee, B., Weideman, H., Takács, G., de Rivaz, P., Crall, J., Sanders, G., Rasul, K., Liu, C., French, G., and Degrave, J. (2015). Lasagne: First release. (Cited on page 45.)

Dong, X., Chen, S., and Pan, S. J. (2017a). Learning to prune deep neural networks via layer-wise optimal brain surgeon. *CoRR*, abs/1705.07565. (Cited on page 8.)

Dong, X., Huang, J., Yang, Y., and Yan, S. (2017b). More is less: A more complicated network with less inference complexity. *CoRR*, abs/1703.08651. (Cited on page 24.)

Drucker, H. and LeCun, Y. (1992). Improving generalization performance using double backpropagation. *IEEE Transaction on Neural Networks*, 3(6):991–997. (Cited on page 66.)

Edwards, H. and Storkey, A. (2017). Towards a neural statistician. In *5th International Conference on Learning Representations (ICLR 2017)*. (Cited on pages 44 and 123.)

Elsken, T., Metzen, J.-H., and Hutter, F. (2017). Simple And Efficient Architecture Search for Convolutional Neural Networks. *ArXiv e-prints*. (Cited on pages 28 and 29.)

Ester, M., Kriegel, H.-P., Sander, J., and Xu, X. (1996). A density-based algorithm for discovering clusters a density-based algorithm for discovering clusters in large spatial databases with noise. In *Proceedings of the Second International Conference on Knowledge Discovery and Data Mining*, KDD'96, pages 226–231. AAAI Press. (Cited on page 50.)

Federici, M., Ullrich, K., and Welling, M. (2017). Improved Bayesian Compression. *ArXiv e-prints*. (Cited on pages 7 and 12.)

Fernando, C., Banarse, D., Reynolds, M., Besse, F., Pfau, D., Jaderberg, M., Lanctot, M., and Wierstra, D. (2016). Convolution by evolution: Differentiable pattern producing networks. *CoRR*, abs/1606.02580. (Cited on page 29.)

Feurer, M., Klein, A., Eggensperger, K., Springenberg, J., Blum, M., and Hutter, F. (2015). Efficient and robust automated machine learning. In Cortes, C., Lawrence, N. D., Lee, D. D., Sugiyama, M., and Garnett, R., editors, *Advances in Neural Information Processing Systems 28*, pages 2962–2970. Curran Associates, Inc. (Cited on page 5.)

Figurnov, M., Ibraimova, A., Vetrov, D. P., and Kohli, P. (2016). PerforatedCNNs: Acceleration through elimination of redundant convolutions. In Lee, D. D., Sugiyama, M., Luxburg, U. V., Guyon, I., and Garnett, R., editors, *Advances in Neural Information Processing Systems 29*, pages 947–955. Curran Associates, Inc. (Cited on pages 13, 32, and 122.)

Flyamer, I., Colin, Xue, Z., Li, A., Vazquez, V., Morshed, N., Neste, C. V., scaine1, and mski_iksm (2018). adjustText. `https://github.com/Phlya/adjustText`. (Cited on page 99.)

Furlanello, T., Lipton, Z., Tschannen, M., Itti, L., and Anandkumar, A. (2018). Born-again neural networks. In *International Conference on Machine Learning*, pages 1602–1611. (Cited on page 77.)

Gal, Y., Hron, J., and Kendall, A. (2017). Concrete Dropout. *ArXiv e-prints*. (Cited on pages 7 and 12.)

Garipov, T., Podoprikhin, D., Novikov, A., and Vetrov, D. P. (2016). Ultimate tensorization: compressing convolutional and FC layers alike. *CoRR*, abs/1611.03214. (Cited on pages 26, 86, 87, 94, and 120.)

Garnelo, M., Rosenbaum, D., Maddison, C. J., Ramalho, T., Saxton, D., Shanahan, M., Whye Teh, Y., Rezende, D. J., and Eslami, S. M. A. (2018). Conditional Neural Processes. *ArXiv e-prints*. (Cited on page 44.)

Gilbert, E. (1955). Theory of shuffling. Technical report, Bell Labs, Murray Hill, New Jersey, U.S. (Cited on page 90.)

Golea, M., Marchand, M., and Hancock, T. R. (1993). On learning μ-perceptron networks with binary weights. In Hanson, S. J., Cowan, J. D., and Giles, C. L., editors, *Advances in Neural Information Processing Systems 5*, pages 591–598. Morgan-Kaufmann. (Cited on page 16.)

Gong, Y., Liu, L., Yang, M., and Bourdev, L. D. (2014). Compressing deep convolutional networks using vector quantization. *CoRR*, abs/1412.6115. (Cited on page 17.)

Goodfellow, I., Bengio, Y., and Courville, A. (2016). *Deep Learning*. MIT Press. http://www.deeplearningbook.org. (Cited on pages 5, 6, 66, 67, and 124.)

Graham, B. and van der Maaten, L. (2017). Submanifold sparse convolutional networks. *CoRR*, abs/1706.01307. (Cited on page 13.)

Guo, Y., Yao, A., and Chen, Y. (2016). Dynamic network surgery for efficient dnns. *CoRR*, abs/1608.04493. (Cited on page 9.)

Gupta, S., Agrawal, A., Gopalakrishnan, K., and Narayanan, P. (2015). Deep learning with limited numerical precision. *CoRR*, abs/1502.02551. (Cited on page 17.)

Guyon, I., Weston, J., Barnhill, S., and Vapnik, V. (2002). Gene selection for cancer classification using support vector machines. *Machine learning*, 46(1-3):389–422. (Cited on page 46.)

Gysel, P., Motamedi, M., and Ghiasi, S. (2016). Hardware-oriented approximation of convolutional neural networks. *CoRR*, abs/1604.03168. (Cited on page 17.)

Ha, D., Dai, A. M., and Le, Q. V. (2016). Hypernetworks. *CoRR*, abs/1609.09106. (Cited on page 25.)

Han, S., Mao, H., and Dally, W. J. (2015). Deep compression: Compressing deep neural network with pruning, trained quantization and huffman coding. *CoRR*, abs/1510.00149. (Cited on pages 2, 7, 10, 11, 12, 18, 30, 33, 69, and 119.)

Hanson, S. J. and Pratt, L. Y. (1989). Comparing biases for minimal network construction with back-propagation. In Touretzky, D. S., editor, *Advances in Neural Information Processing Systems 1*, pages 177–185. Morgan-Kaufmann. (Cited on pages 7 and 9.)

Hassibi, B. and Stork, D. G. (1993). Second order derivatives for network pruning: Optimal brain surgeon. In Hanson, S. J., Cowan, J. D., and Giles, C. L., editors, *Advances in Neural Information Processing Systems 5*, pages 164–171. Morgan-Kaufmann. (Cited on page 9.)

He, K., Zhang, X., Ren, S., and Sun, J. (2016a). Deep residual learning for image recognition. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 770–778. (Cited on pages 5, 6, 15, 20, 21, 22, 23, 26, 27, 33, 62, 68, 69, 77, 81, 84, 96, 101, 106, 126, and 127.)

He, K., Zhang, X., Ren, S., and Sun, J. (2016b). Identity mappings in deep residual networks. In *European Conference on Computer Vision*. (Cited on page 71.)

Hernández-Lobato, J. M. and Adams, R. P. (2015). Probabilistic Backpropagation for Scalable Learning of Bayesian Neural Networks. *ArXiv e-prints*. (Cited on page 30.)

Hinton, G., Vinyals, O., and Dean, J. (2016). Distilling the Knowledge in a Neural Network. *CoRR*, abs/1503.02531. (Cited on pages 28, 63, 64, 65, and 66.)

Hinton, G. E., Srivastava, N., Krizhevsky, A., Sutskever, I., and Salakhutdinov, R. (2012). Improving neural networks by preventing co-adaptation of feature detectors. *CoRR*, abs/1207.0580. (Cited on page 7.)

Hinton, G. E. and van Camp, D. (1993). Keeping neural networks simple by minimizing the description length of the weights. In *Proceedings of the Sixth Annual Conference on Computational Learning Theory*, COLT '93, pages 5–13, New York, NY, USA. ACM. (Cited on pages 6 and 30.)

Hoffer, E., Hubara, I., and Soudry, D. (2018). Fix your classifier: the marginal value of training the last weight layer. In *International Conference on Learning Representations*. (Cited on pages 20 and 84.)

Honkela, A. and Valpola, H. (2004). Variational learning and bits-back coding: an information-theoretic view to bayesian learning. *IEEE Transactions on Neural Networks*, 15(4):800–810. (Cited on page 30.)

Hou, L., Yao, Q., and Kwok, J. T. (2016). Loss-aware binarization of deep networks. *CoRR*, abs/1611.01600. (Cited on page 9.)

Howard, A. G., Zhu, M., Chen, B., Kalenichenko, D., Wang, W., Weyand, T., Andreetto, M., and Adam, H. (2017). Mobilenets: Efficient convolutional neural networks for mobile vision applications. *CoRR*, abs/1704.04861. (Cited on pages 7, 20, 22, 33, 34, 64, 75, 78, 79, 80, 96, 107, 108, 109, 126, and 128.)

Hu, J., Shen, L., and Sun, G. (2017). Squeeze-and-excitation networks. *CoRR*, abs/1709.01507. (Cited on page 23.)

Hu, Z. (2019). torch-dct. `https://github.com/zh217/torch-dct`. Accessed: 14th February 2019. (Cited on page 99.)

Huang, G., Chen, D., Li, T., Wu, F., van der Maaten, L., and Weinberger, K. Q. (2017a). Multi-scale dense convolutional networks for efficient prediction. *CoRR*, abs/1703.09844. (Cited on pages 23, 24, 33, and 122.)

Huang, G., Liu, S., van der Maaten, L., and Weinberger, K. Q. (2017b). CondenseNet: An efficient densenet using learned group convolutions. *CoRR*, abs/1711.09224. (Cited on pages 81, 106, 115, and 127.)

Huang, G., Liu, Z., and Weinberger, K. Q. (2016a). Densely connected convolutional networks. *CoRR*, abs/1608.06993. (Cited on pages 24, 106, 108, and 127.)

Huang, G., Sun, Y., Liu, Z., Sedra, D., and Weinberger, K. Q. (2016b). Deep networks with stochastic depth. *CoRR*, abs/1603.09382. (Cited on pages 23 and 24.)

Hubara, I., Courbariaux, M., Soudry, D., El-Yaniv, R., and Bengio, Y. (2018). Quantized neural networks: Training neural networks with low precision weights and activations. *Journal of Machine Learning Research*, 18(187):1–30. (Cited on page 17.)

Hunter, J. D. (2007). Matplotlib: A 2d graphics environment. *Computing In Science & Engineering*, 9(3):90–95. (Cited on page 99.)

Iandola, F. N., Moskewicz, M. W., Ashraf, K., Han, S., Dally, W. J., and Keutzer, K. (2016). Squeezenet: Alexnet-level accuracy with 50x fewer parameters and <1mb model size. *CoRR*, abs/1602.07360. (Cited on pages 20, 22, 79, 109, 126, and 128.)

Ioannou, Y., Robertson, D., Cipolla, R., and Criminisi, A. (2017). Deep roots: Improving cnn efficiency with hierarchical filter groups. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*. (Cited on page 78.)

Ioannou, Y., Robertson, D. P., Cipolla, R., and Criminisi, A. (2016). Deep roots: Improving CNN efficiency with hierarchical filter groups. *CoRR*, abs/1605.06489. (Cited on page 22.)

Ioffe, S. and Szegedy, C. (2015). Batch normalization: Accelerating deep network training by reducing internal covariate shift. In *International Conference on Machine Learning*, pages 448–456. (Cited on pages 22 and 64.)

Jaderberg, M., Vedaldi, A., and Zisserman, A. (2014). Speeding up Convolutional Neural Networks with Low Rank Expansions. In *British Machine Vision Conference*. (Cited on pages 25 and 86.)

Janisch, J., Pevný, T., and Lisý, V. (2017). Classification with costly features using deep reinforcement learning. *CoRR*, abs/1711.07364. (Cited on pages 1 and 42.)

Jaynes, E., Jaynes, E., Bretthorst, G., and Press, C. U. (2003). *Probability Theory: The Logic of Science*. Cambridge University Press. (Cited on page 38.)

Jin, J., Dundar, A., and Culurciello, E. (2015). Flattened Convolutional Neural Networks for Feedforward Acceleration. In *International Conference on Learning Representations*. (Cited on page 22.)

Judd, P., Albericio, J., Hetherington, T., Aamodt, T. M., Jerger, N. E., and Moshovos, A. (2016a). Proteus: Exploiting numerical precision variability in deep neural networks. In *Proceedings of the 2016 International Conference on Supercomputing*, page 23. ACM. (Cited on page 17.)

Judd, P., Albericio, J., Hetherington, T., Aamodt, T. M., and Moshovos, A. (2016b). Stripes: Bit-serial deep neural network computing. In *Microarchitecture (MICRO), 2016 49th Annual IEEE/ACM International Symposium on*, pages 1–12. IEEE. (Cited on page 17.)

Karaletsos, T. and Rätsch, G. (2015). Automatic Relevance Determination For Deep Generative Models. *ArXiv e-prints*. (Cited on pages 11 and 32.)

Kim, M. and Smaragdis, P. (2016). Bitwise neural networks. *CoRR*, abs/1601.06071. (Cited on page 16.)

Kim, Y.-D., Park, E., Yoo, S., Choi, T., Yang, L., and Shin, D. (2015). Compression of Deep Convolutional Neural Networks for Fast and Low Power Mobile Applications. *ArXiv e-prints*. (Cited on page 25.)

Kingma, D. P. and Ba, J. (2014). Adam: A method for stochastic optimization. *CoRR*, abs/1412.6980. (Cited on pages 48 and 58.)

Kingma, D. P., Salimans, T., and Welling, M. (2015). Variational dropout and the local reparameterization trick. In *Advances in Neural Information Processing Systems*, pages 2575–2583. (Cited on pages 7, 11, 12, and 30.)

Kingma, D. P. and Welling, M. (2014). Auto-encoding variational bayes. In *Proceedings of the Second International Conference on Learning Representations (ICLR 2014)*. (Cited on pages 37, 40, 43, and 59.)

Korattikara Balan, A., Rathod, V., Murphy, K. P., and Welling, M. (2015). Bayesian dark knowledge. In Cortes, C., Lawrence, N. D., Lee, D. D., Sugiyama, M., and Garnett, R., editors, *Advances in Neural Information Processing Systems 28*, pages 3438–3446. Curran Associates, Inc. (Cited on page 28.)

Kossaifi, J., Lipton, Z. C., Khanna, A., Furlanello, T., and Anandkumar, A. (2017). Tensor regression networks. *CoRR*, abs/1707.08308. (Cited on pages 26, 87, 94, 95, 108, and 120.)

Krizhevsky, A. (2009). Learning multiple layers of features from tiny images. Master's thesis, University of Toronto. (Cited on pages 8, 26, 62, 85, 97, 100, 115, 118, and 119.)

Krizhevsky, A., Sutskever, I., and Hinton, G. E. (2012a). Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems*, pages 1097–1105. (Cited on pages 5, 6, 15, 20, 21, 25, 27, 29, 33, 34, and 67.)

Krizhevsky, A., Sutskever, I., and Hinton, G. E. (2012b). Imagenet classification with deep convolutional neural networks. In *Advances in Neural Information Processing Systems*. (Cited on pages 84 and 85.)

Larsson, G., Maire, M., and Shakhnarovich, G. (2016). Fractalnet: Ultra-deep neural networks without residuals. *CoRR*, abs/1605.07648. (Cited on page 24.)

Lavin, A. and Gray, S. (2016). Fast algorithms for convolutional neural networks. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 4013–4021. (Cited on page 27.)

Lawrence, N. D. (2001). Node relevance determination. In Marinaro, M. and Tagliaferri, R., editors, *The XIth Italian Workshop on Neural Networks*. IIASS "Eduardo R. Caianiello", Springer-Verlag. (Cited on page 10.)

Lebedev, V., Ganin, Y., Rakhuba, M., Oseledets, I. V., and Lempitsky, V. S. (2014). Speeding-up convolutional neural networks using fine-tuned cp-decomposition. *CoRR*, abs/1412.6553. (Cited on pages 25 and 86.)

Lebedev, V. and Lempitsky, V. S. (2015). Fast convnets using group-wise brain damage. *CoRR*, abs/1506.02515. (Cited on page 12.)

LeCun, Y., Bengio, Y., and Hinton, G. (2015). Deep learning. *Nature*, 521(7553):436–444. (Cited on page 1.)

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. (Cited on pages 23 and 45.)

LeCun, Y., Denker, J. S., and Solla, S. A. (1990). Optimal brain damage. In Touretzky, D. S., editor, *Advances in Neural Information Processing Systems 2*, pages 598–605. Morgan-Kaufmann. (Cited on pages 8 and 9.)

Li, F. and Liu, B. (2016). Ternary weight networks. *CoRR*, abs/1605.04711. (Cited on page 16.)

Li, H., Kadav, A., Durdanovic, I., Samet, H., and Graf, H. P. (2016). Pruning filters for efficient convnets. *CoRR*, abs/1608.08710. (Cited on page 13.)

Li, L. and Talwalkar, A. (2019). Random search and reproducibility for neural architecture search. *CoRR*, abs/1902.07638. (Cited on page 29.)

Lichman, M. (2013). UCI machine learning repository. (Cited on pages 48 and 123.)

Lin, D. D. and Talathi, S. S. (2016). Overcoming challenges in fixed point training of deep convolutional networks. *CoRR*, abs/1607.02241. (Cited on page 17.)

Lin, M., Chen, Q., and Yan, S. (2013). Network in network. *CoRR*, abs/1312.4400. (Cited on pages 20 and 66.)

Lin, Z., Courbariaux, M., Memisevic, R., and Bengio, Y. (2015). Neural networks with few multiplications. *CoRR*, abs/1510.03009. (Cited on page 15.)

Liu, C., Zoph, B., Shlens, J., Hua, W., Li, L., Fei-Fei, L., Yuille, A. L., Huang, J., and Murphy, K. (2017). Progressive neural architecture search. *CoRR*, abs/1712.00559. (Cited on pages 33, 79, 106, 109, 126, 127, and 128.)

Liu, H., Simonyan, K., and Yang, Y. (2018). DARTS: Differentiable Architecture Search. *ArXiv e-prints*. (Cited on pages 5, 79, 97, 100, 106, 109, 126, 127, and 128.)

Lopez-Paz, D., Bottou, L., Schölkopf, B., and Vapnik, V. (2015). Unifying distillation and privileged information. *ArXiv e-prints*. (Cited on page 28.)

Louizos, C., Ullrich, K., and Welling, M. (2017). Bayesian Compression for Deep Learning. *ArXiv e-prints*. (Cited on pages 7, 12, 17, and 32.)

Lynch, S. (2007). *Introduction to Applied Bayesian Statistics and Estimation for Social Scientists*. Statistics for Social and Behavioral Sciences. Springer New York. (Cited on page 37.)

Ma, Y., Suda, N., Cao, Y., Seo, J.-s., and Vrudhula, S. (2016). Scalable and modularized RTL compilation of convolutional neural networks onto FPGA. In *Field Programmable Logic and Applications (FPL), 2016 26th International Conference on*, pages 1–8. IEEE. (Cited on page 17.)

MacKay, D. J. (1992). A practical bayesian framework for backpropagation networks. *Neural computation*, 4(3):448–472. (Cited on pages 6 and 30.)

MacKay, D. J. et al. (1994). Bayesian nonlinear modeling for the prediction competition. *ASHRAE transactions*, 100(2):1053–1062. (Cited on page 10.)

Mamalet, F. and Garcia, C. (2012). Simplifying convnets for fast learning. In *International Conference on Artificial Neural Networks*, pages 58–65. Springer. (Cited on page 25.)

Mathieu, M., Henaff, M., and LeCun, Y. (2013). Fast training of convolutional networks through ffts. *CoRR*, abs/1312.5851. (Cited on page 26.)

Mellempudi, N., Kundu, A., Mudigere, D., Das, D., Kaul, B., and Dubey, P. (2017). Ternary neural networks with fine-grained quantization. *CoRR*, abs/1705.01462. (Cited on pages 16 and 33.)

Merolla, P., Appuswamy, R., Arthur, J. V., Esser, S. K., and Modha, D. S. (2016). Deep neural networks are robust to weight binarization and other non-linear distortions. *CoRR*, abs/1606.01981. (Cited on page 30.)

Miyashita, D., Lee, E. H., and Murmann, B. (2016). Convolutional neural networks using logarithmic data representation. *CoRR*, abs/1603.01025. (Cited on page 17.)

Moczulski, M., Denil, M., Appleyard, J., and de Freitas, N. (2015). ACDC: A structured efficient linear layer. *CoRR*, abs/1511.05946. (Cited on pages 27, 33, 34, 87, 88, 90, 91, 107, 108, 110, and 116.)

Molchanov, D., Ashukha, A., and Vetrov, D. (2017). Variational Dropout Sparsifies Deep Neural Networks. *ArXiv e-prints*. (Cited on pages 7, 11, and 12.)

Molchanov, P., Tyree, S., Karras, T., Aila, T., and Kautz, J. (2016). Pruning convolutional neural networks for resource efficient transfer learning. *CoRR*, abs/1611.06440. (Cited on page 9.)

Moons, B., De Brabandere, B., Van Gool, L., and Verhelst, M. (2016). Energy-efficient convnets through approximate computing. In *Applications of Computer Vision (WACV), 2016 IEEE Winter Conference on*, pages 1–8. IEEE. (Cited on page 17.)

Moons, B. and Verhelst, M. (2016). A 0.3–2.6 tops/w precision-scalable processor for real-time large-scale convnets. In *VLSI Circuits (VLSI-Circuits), 2016 IEEE Symposium on*, pages 1–2. IEEE. (Cited on page 17.)

Murphy, K. P. (2012). *Machine Learning: A Probabilistic Perspective*. The MIT Press, London. (Cited on pages 5, 6, 37, 38, and 97.)

Nalisnick, E., Anandkumar, A., and Smyth, P. (2015). A Scale Mixture Perspective of Multiplicative Noise in Neural Networks. *ArXiv e-prints*. (Cited on page 11.)

Nan, F. and Saligrama, V. (2017). Adaptive classification for prediction under a budget. In Guyon, I., Luxburg, U. V., Bengio, S., Wallach, H., Fergus, R., Vishwanathan, S., and Garnett, R., editors, *Advances in Neural Information Processing Systems 30*, pages 4727–4737. Curran Associates, Inc. (Cited on page 42.)

Nan, F., Wang, J., and Saligrama, V. (2015). Feature-budgeted random forest. In Bach, F. and Blei, D., editors, *Proceedings of the 32nd International Conference on Machine Learning*, volume 37 of *Proceedings of Machine Learning Research*, pages 1983–1991, Lille, France. PMLR. (Cited on page 42.)

Nan, F., Wang, J., and Saligrama, V. (2016). Pruning random forests for prediction on a budget. In Lee, D. D., Sugiyama, M., Luxburg, U. V., Guyon, I., and Garnett, R., editors, *Advances in Neural Information Processing Systems 29*, pages 2334–2342. Curran Associates, Inc. (Cited on page 42.)

Neal, R. M. (1995). *Bayesian learning for neural networks*. PhD thesis, University of Toronto. (Cited on page 10.)

Neklyudov, K., Molchanov, D., Ashukha, A., and Vetrov, D. (2017). Structured Bayesian Pruning via Log-Normal Multiplicative Noise. *ArXiv e-prints*. (Cited on page 11.)

Novikov, A., Podoprikhin, D., Osokin, A., and Vetrov, D. P. (2015). Tensorizing neural networks. *CoRR*, abs/1509.06569. (Cited on pages 26, 87, 94, and 108.)

Nvidia (2015). Tesla K40 GPU accelerator. `http://www.nvidia.com/content/PDF/kepler/Tesla-K40-PCIe-Passive-Board-Spec-BD-06902-001_v05.pdf`. Accessed: 11th December 2015. (Cited on page 117.)

O'Connor, P. and Welling, M. (2016). Sigma delta quantized networks. *CoRR*, abs/1611.02024. (Cited on page 24.)

Oseledets, I. V. (2011). Tensor-train decomposition. *SIAM Journal on Scientific Computing*, 33(5):2295–2317. (Cited on pages 86, 93, and 94.)

Ott, J., Lin, Z., Zhang, Y., Liu, S., and Bengio, Y. (2016). Recurrent neural networks with limited numerical precision. *CoRR*, abs/1611.07065. (Cited on page 16.)

Page, D. (2019). cifar10-fast. `https://github.com/davidcpage/cifar10-fast`. Accessed: 14th February 2019. (Cited on page 100.)

Parisotto, E., Ba, L. J., and Salakhutdinov, R. (2015). Actor-mimic: Deep multitask and transfer reinforcement learning. *CoRR*, abs/1511.06342. (Cited on page 28.)

Paszke, A., Gross, S., Chintala, S., and Chanan, G. (2017). PyTorch: Tensors and dynamic neural networks in Python with strong GPU acceleration. https://github.com/pytorch/pytorch. Accessed: 31st October 2017. (Cited on pages 45 and 99.)

Paszke, A., Gross, S., Chintala, S., and Chanan, G. (2019). torchvision.models PyTorch master documentation. https://pytorch.org/docs/stable/torchvision/models.html#torchvision.models.wide_resnet50_2. Accessed: 19th August 2019. (Cited on page 101.)

Peter, S., Diego, F., Hamprecht, F. A., and Nadler, B. (2017). Cost efficient gradient boosting. In Guyon, I., Luxburg, U. V., Bengio, S., Wallach, H., Fergus, R., Vishwanathan, S., and Garnett, R., editors, *Advances in Neural Information Processing Systems 30*, pages 1551–1561. Curran Associates, Inc. (Cited on page 42.)

Pham, H., Guan, M. Y., Zoph, B., Le, Q. V., and Dean, J. (2018). Efficient neural architecture search via parameter sharing. *CoRR*, abs/1802.03268. (Cited on page 29.)

Pitt, L. and Valiant, L. G. (1988). Computational limitations on learning from examples. *Journal of the Association for Computing Machinery*, 35(4):965–984. (Cited on page 16.)

Prashnani, E., Cai, H., Mostofi, Y., and Sen, P. (2018). PieAPP: Perceptual image-error assessment through pairwise preference. In *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. (Cited on pages 53, 54, 55, 57, 59, and 124.)

Price, R. (1958). A useful theorem for nonlinear devices having Gaussian inputs. *IRE Transactions on Information Theory*, IT-4:69–72. (Cited on page 37.)

Rakhuba, M. V. and Oseledets, I. V. (2014). Fast multidimensional convolution in low-rank formats via cross approximation. *ArXiv e-prints*. (Cited on page 26.)

Ranganath, R., Gerrish, S., and Blei, D. M. (2013). Black Box Variational Inference. *arXiv e-prints*, page arXiv:1401.0118. (Cited on page 59.)

Rastegari, M., Ordonez, V., Redmon, J., and Farhadi, A. (2016). XNOR-net: Imagenet classification using binary convolutional neural networks. *CoRR*, abs/1603.05279. (Cited on pages 15 and 32.)

Reagen, B., Whatmough, P., Adolf, R., Rama, S., Lee, H., Lee, S. K., Hernández-Lobato, J. M., Wei, G.-Y., and Brooks, D. (2016). Minerva: Enabling low-power, highly-accurate deep neural network accelerators. In *Proceedings of the 43rd International Symposium on Computer Architecture*, pages 267–278. IEEE Press. (Cited on page 13.)

Real, E., Moore, S., Selle, A., Saxena, S., Suematsu, Y. L., Le, Q. V., and Kurakin, A. (2017). Large-scale evolution of image classifiers. *CoRR*, abs/1703.01041. (Cited on pages 6, 29, 79, 106, 109, 126, 127, and 128.)

Richman, O. and Mannor, S. (2016). How to allocate resources for features acquisition? *CoRR*, abs/1607.02763. (Cited on pages 1, 39, 40, 42, and 59.)

Rigamonti, R., Sironi, A., Lepetit, V., and Fua, P. (2013). Learning separable filters. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 2754–2761. (Cited on page 25.)

Romera, E., Álvarez, J. M., Bergasa, L. M., and Arroyo, R. (2017a). Efficient convnet for real-time semantic segmentation. *IEEE Intelligent Vehicles Symposium*. (Cited on page 80.)

Romera, E., Álvarez, J. M., Bergasa, L. M., and Arroyo, R. (2017b). ERFNet: Efficient residual factorized convnet for real-time semantic segmentation. *IEEE Transactions on Intelligent Transportation Systems*. (Cited on pages 80 and 81.)

Romero, A., Ballas, N., Kahou, S. E., Chassang, A., Gatta, C., and Bengio, Y. (2014). Fitnets: Hints for thin deep nets. *CoRR*, abs/1412.6550. (Cited on pages 28 and 61.)

Saade, A., Caltagirone, F., Carron, I., Daudet, L., Drémeau, A., Gigan, S., and Krzakala, F. (2015). Random projections through multiple optical scattering: Approximating kernels at the speed of light. *CoRR*, abs/1510.06664. (Cited on pages 116 and 120.)

Salimans, T. and Knowles, D. A. (2013). Fixed-form variational posterior approximation through stochastic linear regression. *Bayesian Analysis*, 8(4):837–882. (Cited on page 37.)

Scardapane, S., Comminiello, D., Hussain, A., and Uncini, A. (2016). Group Sparse Regularization for Deep Neural Networks. *ArXiv e-prints*. (Cited on page 13.)

Schulman, J., Heess, N., Weber, T., and Abbeel, P. (2015). Gradient estimation using stochastic computation graphs. In Cortes, C., Lawrence, N. D., Lee, D. D., Sugiyama, M., and Garnett, R., editors, *Advances in Neural Information Processing Systems 28*, pages 3528–3536. Curran Associates, Inc. (Cited on pages 42, 43, 59, and 123.)

Shwartz-Ziv, R. and Tishby, N. (2017). Opening the black box of deep neural networks via information. *CoRR*, abs/1703.00810. (Cited on pages 6, 31, and 122.)

Sifre, L. (2014). *Rigid-Motion Scattering for Image classification*. PhD thesis, École Polytechnique. (Cited on page 22.)

Simonyan, K. and Zisserman, A. (2014). Very deep convolutional networks for large-scale image recognition. *CoRR*, abs/1409.1556. (Cited on pages 9, 18, 19, 21, 25, 26, and 33.)

Snell, J., Swersky, K., and Zemel, R. S. (2017). Prototypical networks for few-shot learning. *CoRR*, abs/1703.05175. (Cited on page 44.)

Soudry, D., Hubara, I., and Meir, R. (2014). Expectation backpropagation: Parameter-free training of multilayer neural networks with continuous or discrete weights. In Ghahramani, Z., Welling, M., Cortes, C., Lawrence, N., and Weinberger, K., editors, *Advances in Neural Information Processing Systems 27*, pages 963–971. Curran Associates, Inc. (Cited on pages 15, 16, and 33.)

Springenberg, J. T., Dosovitskiy, A., Brox, T., and Riedmiller, M. A. (2014). Striving for simplicity: The all convolutional net. *CoRR*, abs/1412.6806. (Cited on pages 19, 21, 26, 67, and 122.)

Srinivas, S. and Babu, R. V. (2015). Data-free parameter pruning for deep neural networks. *CoRR*, abs/1507.06149. (Cited on page 9.)

Srivastava, N., Hinton, G., Krizhevsky, A., Sutskever, I., and Salakhutdinov, R. (2014). Dropout: A simple way to prevent neural networks from overfitting. *Journal of Machine Learning Research*, 15:1929–1958. (Cited on pages 6 and 7.)

Stevens, J.-L. R., Rudiger, P., and Bednar, J. A. (2015). Holoviews: Building complex visualizations easily for reproducible science. In *SciPy Conference Proceedings*. (Cited on pages 45 and 99.)

Su, J., Li, J., Bhattacharjee, B., and Huang, F. (2018). Tensorial Neural Networks: Generalization of Neural Networks and Application to Model Compression. *arXiv*

e-prints, page arXiv:1805.10352. (Cited on page 86.)

Sutskever, I., Martens, J., Dahl, G., and Hinton, G. (2013). On the importance of initialization and momentum in deep learning. In *International conference on machine learning*, pages 1139–1147. (Cited on page 100.)

Sze, V., Chen, Y., Yang, T., and Emer, J. S. (2017). Efficient processing of deep neural networks: A tutorial and survey. *CoRR*, abs/1703.09039. (Cited on page 32.)

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. (Cited on pages 15, 20, 32, 33, 79, 106, 108, 109, 126, 127, and 128.)

Tai, C., Xiao, T., Wang, X., and E, W. (2015). Convolutional neural networks with low-rank regularization. *CoRR*, abs/1511.06067. (Cited on page 25.)

The GPyOpt authors (2016). GPyOpt: A bayesian optimization framework in python. http://github.com/SheffieldML/GPyOpt. (Cited on pages 45 and 46.)

Theis, L., Korshunova, I., Tejani, A., and Huszár, F. (2018). Faster gaze prediction with dense networks and Fisher pruning. *ArXiv e-prints*. (Cited on pages 9 and 32.)

Ullrich, K., Meeds, E., and Welling, M. (2017). Soft Weight-Sharing for Neural Network Compression. *ArXiv e-prints*. (Cited on page 30.)

Urban, G., Geras, K. J., Kahou, S. E., Aslan, O., Wang, S., Caruana, R., Mohamed, A., Philipose, M., and Richardson, M. (2017). Do Deep Convolutional Nets Really Need to be Deep and Convolutional? In *International Conference on Learning Representations*. (Cited on pages 28 and 61.)

Vapnik, V. and Izmailov, R. (2015). Learning using privileged information: Similarity control and knowledge transfer. *Journal of Machine Learning Research*, 16:2023–2049. (Cited on page 28.)

Venkatesh, G., Nurvitadhi, E., and Marr, D. (2016). Accelerating deep convolutional networks using low-precision and sparsity. *CoRR*, abs/1610.00324. (Cited on page 17.)

Wallace, G. K. (1991). The jpeg still picture compression standard. *Communications of the Association for Computing Machinery*, 34(4):30–44. (Cited on pages 2, 51, 52, 53, and 59.)

Wang, M., Liu, B., and Foroosh, H. (2016). Factorized convolutional neural networks. *CoRR*, abs/1608.04337. (Cited on page 22.)

Welch, T. A. (1984). A technique for high-performance data compression. *Computer*, 17(6):8–19. (Cited on pages 14 and 18.)

Wen, W., Wu, C., Wang, Y., Chen, Y., and Li, H. (2016). Learning structured sparsity in deep neural networks. *CoRR*, abs/1608.03665. (Cited on page 13.)

Wen, W., Xu, C., Wu, C., Wang, Y., Chen, Y., and Li, H. (2017). Coordinating filters for faster deep neural networks. *CoRR*, abs/1703.09746. (Cited on page 86.)

Wikimedia (2019). File:dctjpeg.png. https://commons.wikimedia.org/wiki/File:Dctjpeg.png. Accessed: 27th February 2019. (Cited on pages 51 and 124.)

Wintenby, J. and Krishnamurthy, V. (2006). Hierarchical resource management in adaptive airborne surveillance radars. *IEEE Transactions on Aerospace and Electronic systems*, 42(2):401–420. (Cited on page 37.)

Wu, J., Leng, C., Wang, Y., Hu, Q., and Cheng, J. (2015). Quantized convolutional neural networks for mobile devices. *CoRR*, abs/1512.06473. (Cited on page 17.)

Xie, S., Girshick, R., Dollár, P., Tu, Z., and He, K. (2017). Aggregated residual transformations for deep neural networks. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*. (Cited on page 64.)

Xu, Z., Weinberger, K., and Chapelle, O. (2012). The Greedy Miser: Learning under Test-time Budgets. *ArXiv e-prints*. (Cited on page 42.)

Yamada, Y., Iwamura, M., and Kise, K. (2016). Deep pyramidal residual networks with separated stochastic depth. *CoRR*, abs/1612.01230. (Cited on page 23.)

Yang, T., Chen, Y., and Sze, V. (2016). Designing energy-efficient convolutional neural networks using energy-aware pruning. *CoRR*, abs/1611.05128. (Cited on page 13.)

Yang, Z., Moczulski, M., Denil, M., de Freitas, N., Smola, A., Song, L., and Wang, Z. (2015). Deep fried convnets. In *The IEEE International Conference on Computer Vision (ICCV)*. (Cited on pages 27 and 69.)

Yosinski, J., Clune, J., Bengio, Y., and Lipson, H. (2014). How transferable are features in deep neural networks? In *Advances in neural information processing systems*, pages 3320–3328. (Cited on page 30.)

Yu, F. and Koltun, V. (2016). Multi-Scale Context Aggregation by Dilated Convolutions. In *International Conference on Learning Representations*. (Cited on page 72.)

Zagoruyko, S. and Komodakis, N. (2016). Wide residual networks. In *British Machine Vision Conference*. (Cited on pages 71, 76, 97, 100, 101, 108, and 125.)

Zagoruyko, S. and Komodakis, N. (2017). Paying More Attention to Attention: Improving the Performance of Convolutional Neural Networks via Attention Transfer. In *International Conference on Learning Representations*. (Cited on pages 5, 28, 63, 64, 65, 66, 72, 75, 77, 78, 97, 100, and 124.)

Zhai, S., Cheng, Y., Lu, W., and Zhang, Z. (2016). Doubly convolutional neural networks. *CoRR*, abs/1610.09716. (Cited on page 23.)

Zhang, X., Zhou, X., Lin, M., and Sun, J. (2017a). ShuffleNet: An Extremely Efficient Convolutional Neural Network for Mobile Devices. *ArXiv e-prints*. (Cited on pages 22, 33, 79, 80, 90, 95, 96, 108, 109, 126, and 128.)

Zhang, X., Zou, J., He, K., and Sun, J. (2015). Accelerating very deep convolutional networks for classification and detection. *CoRR*, abs/1505.06798. (Cited on page 26.)

Zhang, Y., Xiang, T., Hospedales, T. M., and Lu, H. (2017b). Deep Mutual Learning. *ArXiv e-prints*. (Cited on page 28.)

Zhou, A., Yao, A., Guo, Y., Xu, L., and Chen, Y. (2017). Incremental network quantization: Towards lossless cnns with low-precision weights. *CoRR*, abs/1702.03044. (Cited on pages 17 and 33.)

Zhou, S., Ni, Z., Zhou, X., Wen, H., Wu, Y., and Zou, Y. (2016). Dorefa-net: Training low bitwidth convolutional neural networks with low bitwidth gradients. *CoRR*, abs/1606.06160. (Cited on page 17.)

Zhu, C., Han, S., Mao, H., and Dally, W. J. (2016). Trained ternary quantization. *CoRR*, abs/1612.01064. (Cited on page 16.)

Zoph, B. and Le, Q. V. (2016). Neural architecture search with reinforcement learning. *CoRR*, abs/1611.01578. (Cited on pages 5, 28, 29, and 117.)

Zoph, B., Vasudevan, V., Shlens, J., and Le, Q. V. (2017). Learning transferable architectures for scalable image recognition. *CoRR*, abs/1707.07012. (Cited on pages 78, 79, 88, 106, 109, 126, 127, and 128.)