School of Computer Science and Engineering

Faculty of Engineering

The University of New South Wales

Resource Efficient Deep Neural Networks For Nanosatellites

by

Adam Florence

Thesis submitted as a requirement for the degree of

Bachelor of Engineering in Computer Engineering

Submitted: November 2021 Student ID: z5117268

Supervisor: A/Prof. Gustavo Batista

Abstract

In this thesis, we experimentally investigate the problem of running resource-intensive deep neural networks in edge environments lacking processing power and energy availability, specifically on-board nanosatellites. In doing so, we select a data-set of satellite imagery for object detection, and employ an established object detection neural network to use a base model.

With this base model, we train, evaluate and compare many of the most promising techniques that have been proposed to enable deep neural networks to be smaller and more efficient with minimal loss of accuracy. To test efficiency, we simulate an edge environment to measure realistic energy usage. Through testing we are able to evaluate the trade-off between accuracy and energy efficiency achieved by implementing these techniques.

Key Terms: deep neural networks, resource-efficient neural networks, neural network pruning, neural network quantization, nanosatellites.

Acknowledgements

I would like to thank my supervisor, Gustavo Batista, without whom this thesis would not be possible. Thank you for guiding my work, taking the time to meet with me every week, helping me find the best reading material, wiring our hardware, always giving great feedback and advice, bringing me onto this project, and providing invaluable expertise.

Additionally, I would like to thank William Hales for providing crucial guidance on hardware issues. As well as Claude Sammut for being my thesis assessor.

Our entire project has originated from SmartSat CRC. For this I am deeply grateful to have been given the opportunity to work on such an interesting project.

This thesis has been crucially assisted by the body of research on the topic, especially Roth et al. [RSZ+20] for compiling much of the required reading material. Furthermore, many of these researchers have made public the code they used in their research, without which this kind of research would not be an option. Unfortunately, I cannot thank every author cited in this work individually despite their contribution.

Finally, I would like to thank my girlfriend, Sabrina, and my Dad, Rick Florence, for always being available for support and proof reading.

Abbreviations

BN Batch Normalisation

BNN Binarised Neural Network

CNN Convolutional Neural Network

DNN Deep Neural Network

DRAM Dynamic Random Access Memory

FLOP Floating Point Operation

GB Gigabyte

GPU Graphics Processing Unit

GUI Graphical User Interface

IoT Internet of Things

IOU Intersection Over Union

ISR Intelligence, Surveillance and Reconnaissance

mAP Mean Average Precision

MSE Mean Squared Error

RGB Red Green Blue

YOLO You Only Look Once

Contents

Chapter 1

Introduction

With the beginning of the personal computer industry in the late 1970's, the power of the computer would now be able to be harnessed on a level never seen before. Computer engineers were making transistors smaller at an exponential rate, such that where previously an entire room had to be dedicated to a computer, one could now fit at a desk. Within decades computers became ubiquitous, bringing forth the digital age of humanity, and forever changing the way people interact with the world.

Recent years are showing that machine learning technology, specifically deep neural networks (DNNs), is progressing in a similar way. Today, the main domain for utilising machine learning is on large rack-mounted computer systems dedicated to large scale data processing. This is due to the fact that currently, useful DNNs are generally very large in size, and thus require a lot of power intensive processing. But with the fast progress being made in machine learning applications such as natural language processing and computer vision, there is more and more reason to make DNNs workable on smaller, more portable hardware such as mobile phones, internet of things (IoT) devices, as well as nanosatellites (very small satellites weighing between 1kg and 10kg). By enabling this sort of intelligence at everyone's fingertips, machine learning could become omnipresent, and transform human interaction with the world just like the personal computer did.

Nanosatellite technology, as a specific use-case for running resource efficient DNNs, is what we are researching in this thesis. Present-day intelligence, surveillance and reconnaissance (ISR) satellites are generally deployed to collect Earth observation imagery and simply forward this unprocessed data imagery data to a ground station. The ground station is then responsible for processing this data. Since a direct line of sight between a satellite and its receiver on the ground is required for communication, data throughput is limited. This bottleneck introduces a delay between the collection of the data on the satellite and the use of the information on the ground.

A solution to this delay is to process the imagery data on-board the ISR satellite. This processing will very often come in the form of filtering out useful information from the masses of data collected. Real world examples of this filtering are cloud removal (removal of images of cloud), and ship detection (removal of images without ships). By being done on the satellite, much less data needs to be transmitted to the ground and the pressure on the communication bottleneck is eased. This enables comparatively fast communication from space to the field operatives who use the imagery.

The reason that contemporary ISR systems do not utilise on-board processing, is that this kind of large-scale image data processing will require resource intensive DNNs for classification (between imagery worth keeping and imagery not worth keeping). Typically, running DNNs will require large graphics processing units (GPUs). However, with the physical space and power allocated to a nanosatellite, only edge computing hardware can be used. The challenge that we are tackling in this thesis is how do we train DNNs that are power-efficient and small enough to overcome the constraints of this kind of hardware, while preserving accuracy.

Presently, the state of deep neural network technology is such that they are very useful in many applications, especially computer vision, but are very intensive in terms of both computational and memory requirements. As an example, DeepFace, the facial recognition system, uses over 120 million parameters in it's DNN [TYRW14]. Structures of this size do not fit in on-chip memory and require much more costly dynamic random access memory (DRAM) accesses. At this scale, DNNs are unfeasible for devices

without large dedicated GPUs.

In order to create a usable DNN to run in the conditions of a nanosatellite, the network will need to be configured to require as little power and time to run as possible. We need to minimise the floating point operations (FLOPs) required, so time to inference is reduced. Also, we need to minimise the memory usage, since memory accesses are responsible for the majority of power usage of a computer. In essence this boils down to reducing the size of the DNN (in terms of the structure of the network and the total number of parameters) however we can without losing accuracy.

The goal of this thesis is to provide experimental results that can be used to find a workable solution for developing and running a resource-efficient DNN in an environment with limited available power. The DNN use-case we are dealing with is filtering out useful data, and we are specifically dealing with ship detection. Although we will be investigating solutions for image filtering on nanosatellites, the techniques being investigated and results collected will apply generally to applications requiring resource-efficient DNNs, including mobile phones and IoT devices.

As there is so much advantage to gain for making DNNs more efficient, there has been an abundance of research on the topic over the last decade. Many promising techniques have been suggested to deal with this problem. We select some of the most promising and suitable techniques for evaluation in terms of our use-case.

We will implement each of these techniques individually using the same data set and base network. For each technique we will train a DNN using our base network as a starting point. Each DNN will then be evaluated for both accuracy and energy efficiency. With these data we can examine which solutions provide the best trade-off between accuracy and energy efficiency.

Chapter 2 reviews the current literature on resource-efficiency techniques for DNNs. Chapter 3 describes the approach and methodology we undertake to test some of these techniques. Chapter 4 outlines the results we have obtained. Chapter 5 gives a conclusion and discusses our results and what we have learned.

Chapter 2

Literature Review

Production level deep neural networks are generally huge in terms of the number of parameters and the computation required to achieve an inference. So naturally, researchers have recognised this weakness in what is otherwise an extremely useful technology and have attempted to solve it. Thus, there is now a plethora of proposed techniques to help alleviate the resource demands of a DNN while maintaining a high level of accuracy.

With the success of so many different ideas in this space, beginning with *Optimal Brain Damage* from LeCun et al. (1989) [LDS89], it is undeniable that traditional DNNs exhibit a massive degree of redundancy in their structure [DSD+13].

Over the last decade, we have seen researchers examine this problem from many different angles, seeking to find the best techniques to exploit the over-paramaterisation of DNNs for resource-efficiency gains. Roth et al. (2020) [RSZ+20] review the current state of the literature, and broadly group the techniques into three categories with some overlap: pruning; quantisation; and structural efficiency.

The first category is the techniques that attempt to *prune* neural networks. That is, removing non-essential parts of a neural network to sparsify or shrink it for efficiency gains. Neural network pruning further subdivides into *structured* and *unstructured*

pruning. Unstructured pruning methods will work at the scale of individual network weights, and set them to zero with the aim of creating a sparse network consisting on the most salient weights. Structured pruning methods instead remove whole structures of a network (such as convolutional channels or neurons) in order to shrink the network into a smaller network consisting of the most salient structures.

A key difference here is that unstructured pruning techniques result in a sparse network, and in order to allow the efficiency gains, the tensors need to be structured in memory as a sparse tensor, introducing some computational overhead. Whereas, the structured pruning techniques simply results in a network where the dimensions of the layers are decreased, and efficiency gains are immediate. In this review we will look at weight-level pruning (unstructured) and network slimming (structured).

The second category techniques attempt to gain efficiency by using other data-types to represent the weights of the network. While, a DNN's weights are generally represented by 32-bit floating points, quantisation methods instead will use either lower bit-width values (e.g. 8-bit floating points) or types for which operations have a lower energy cost (e.g. 32-bit integers).

In most cases, the quantisation method will use a type with both lower bit-width and less costly operations (e.g. 8-bit integers), to achieve a lower expense both in terms of memory and computation. In this review we will look at quantisation as well as post-training quantisation and quantisation-aware training.

The third category is the broadest, consisting of methods that intelligently find a more efficient structure for a DNN in a way that still maintains the knowledge contained in the larger, less efficient network. In some cases, the method will aim to find better efficiencies in the domain of neural network architecture. Other methods will transform the types of data structures that represent the DNN in order to make memory usage gains. In this review we will look at knowledge distillation, weight sharing, retraining from scratch and neural architecture search.

2.1 Weight-Level Pruning

Weight-level pruning (unstructured pruning) techniques will, for the most part, vary based on their method of selecting which weights in a neural network should be pruned, and which kept. They all have the goal of producing a neural network that requires as few FLOPs and DRAM accesses as possible while maintaining the accuracy of the original inefficient network.

Earlier methods, such as *Optimal Brain Damage*, LeCun et al. (1989) [LDS89], and *Optimal Brain Surgery*, Hassibi et al. (1993) [HS93], used a weight selection approach based on each weight's impact on the network's loss function. Estimating these impacts requires costly and complicated algorithms to produce covariance matrices, which is feasible in smaller neural network, but scales poorly. So although it can produce better results than magnitude based methods, it is not feasible in the deep neural networks that we are working on here.

The more recent unstructured pruning techniques will usually fall back on simpler to compute, magnitude based weight threshold for pruning, and still manage to achieve great results.

Han et al. (2015) [HPTD15] proposes an iterative weight-based pruning approach for DNNs. Starting with a trained, over-parameterised network, they will first run their pruning algorithm, then retrain their resulting network. This way, the pruning stage increases efficiency, and the retraining stage allows the network to recover most of its knowledge loss from the pruning. The pruning threshold used is a constant parameter multiplied by the standard deviation of the weights in the layer.

This *prune then retrain* method can be repeated multiple times to maximise efficiency. For each retraining stage, Han et al. (2015) [HPTD15] notice that as the increasingly sparse network has selected the most useful weights, there is less variance in the prediction, and thus over-fitting decreases. Since dropout is used to prevent over-fitting, the dropout-ratio should be reduced. Furthermore, the researchers tested both L1 and L2 regularisation, finding that L2 gives a better accuracy after the retraining stage. With

this, they reduce the size of AlexNet [KSH12] by a factor of 9 without accuracy loss.

Guo et al. [GYC16] improved this method by creating a structure during training that allowed for pruned weights to be brought back in subsequent iterations. They use a binary mask structure, $t_{l,i,j} \in \{0, 1\}$, to represent the pruned state of each weight in the network. This way if a weight comes back above the threshold, we can *unprune* it by setting $t_{l,i,j} = 1$. This method reduced the size of AlexNet by a factor of 17.7 without accuracy loss.

Once the necessary weights have been pruned, we can then convert the dense matrix format into a sparse matrix format. This requires much less memory to store, and thus means far fewer memory accesses. However, this does means additional overhead in converting the sparse matrix into a dense one, which is required to run the network.

2.2 Network Slimming

Structured pruning techniques are usually designed to prune either neurons or convolutional channels. Since DNNs are more often than not convolutional neural networks (CNNs), and this thesis is dealing specifically with computer vision (which uses CNNs), the channel pruning approaches will be more important to us.

Network Slimming is a channel pruning technique for CNNs proposed by Liu et al. (2017) [LLS+17]. The genius of this technique is that it uses structures already built into DNNs to calculate the pruning threshold.

Batch normalisation (BN) [IS15] is a technique used to improve the performance of a CNN. Each convolutional channel's activations are transformed such that they are normally distributed. To achieve this, the BN layer must maintain scale and shift parameters for each channel in the network. Thus, Network Slimming will use these scaling factors that already exist in the network as a pruning threshold.

Liu et al. (2017) [LLS+17] begin with an over-parameterised base network. They add an

L1 regularisation on the scaling factors in each BN layer. This will push the BN scaling factors towards zero when they train the network with the regularisation built into the loss function. They can then effectively rank the importance of each convolutional channel based on it's scaling factor, and prune the channels with a scaling factor below a certain level.

Similar to the weight-level pruning techniques, after pruning we retrain the pruned network. The *prune then retrain* method can be repeated to further prune the network. This network slimming method pruned 70% of the channels of VGGNet [SZ15] without accuracy loss. Since pruning a channel means not needing the input and output parameters of the channel, this 70% reduction means a parameter reduction by a factor 10.

Unlike with unstructured pruning, there is no overhead required to yield the efficiency gains from the pruning. By removing channels we effectively shrink the architecture of the neural network, by removing entire matrices (unlike unstructured pruning, which is designed to sparsify a network).

2.3 Quantisation

Quantisation methods are used in DNNs to change the way each weight and activation in the network is represented. Typically, weights and activations come in the form of a 32-bit floating point as a data type. When it comes to reducing the cost of operating a DNN in terms of memory and computation, changing these parameters to a lower bit-width form is a useful technique. Research has shown that quantising a DNN to integer, boolean or ternary form can embody the knowledge space held in the network without underfitting, as it effectively exploits the redundancy within the over-parameterised network.

The key complication associated with quantised neural networks is how to train them. A discrete-valued neural network cannot simply use gradient descent like we can with

floating-points. In order to implement a quantised network, we also need an effective way of optimising the network in training.

Although quantisation generally implies changing the data-type of the weights to integer, boolean or ternary values, we can also make a DNN more efficient by simply using lower bit-width float values.

Integer quantisation will use 4, 8 or 16-bit integers as parameters. Clearly, we get a gain in memory usage by using lower bit-width values, but furthermore, the energy cost of operating on integers is much better than floating points. For example, the energy cost of a multiplication for 32-bit floating points is 18 times that of 8-bit integers, and 30 times for addition [SCYE17].

Wu et al. (2018) [WLCS18] proposes a framework called *WAGE* in which weights, activations, gradients and errors are constrained to low bit-width integers. Here a gradient is an individual weight's contribution to the loss function, and an error is an individual activation's contribution.

The WAGE framework sets weights, activations, gradients and errors to 8-bit integers, and then weights are further quantised into 2-bit ternary operators, $w \in \{-1, 0, -1\}$, during inference.

Due to the fact that multiplying two 8-bit integers give a 16-bit integer, propagation through the network requires a quantisation after each multiply accumulate cycle. This applies both to forward and back propagation. The quantisation function is over-summarised as follows, where k is the required bit-width and x is the input:

$$Q(x,k) = Clip\{\sigma(k) \times round[x/\sigma(k)], -1 + \sigma(k), 1 - \sigma(k)\}, \sigma(k) = 2^{1-k}, k \in \mathbb{N}_+$$

In order to prevent our low bit-width gradients from stalling due to rounding to zero every time, a method called stochastic rounding is employed. Stochastic rounding will round up or down to its quantised value probabilistically in proportion to the value's distance to its nearest quantised value.

The WAGE framework ultimately managed to achieve an accuracy of 99.6% on the

MNIST data set, and 98.08% on the SVHN [GBI+14] data set. These numbers represent accuracy at the levels we see with state-of-the-art DNNs with no quantisation or energy constraints.

2.4 Post-Training Quantisation

Most of the literature on the topic of quantisation will not only employ some sort of training on the quantised model, but also assume that this training is crucial to the success of the quantised model.

Post-training quantisation shows that this assumption is not necessarily true. Although training is very powerful in general, and helps to offset the loss of accuracy of a DNN due to quantisation, it is not always doable due to real-world constraints, nor is it always necessary.

Banner et al. (2019) [BNHS19] suggest that quantising the weights of a network after the training stage, does not requiring fine-tuning. This is achieved by creating methods to minimise the errors introduced during the quantisation process.

Most quantisation methods will quantise by uniformly dividing entire distribution of weights in a layer. They propose a method called *Analytical Clipping for Integer Quantisation* where all weights outside the range $[-a, a]$ are clipped so $|w| = a$. At this point the weights are quantised uniformly across the range $[-a, a]$. This does mean that the values are often quite distorted from the original value, however since the distribution range is smaller, the rounding error is reduced for the majority of the values. As a result, the final error is smaller.

Banner et al. (2019) [BNHS19] also introduce the idea of per-channel bit allocation, in which a certain average number of bits is allocated among the channels in a convolutional layer for different quantisation levels. By allocating each channel's parameters a number of bits in proportion to the a value of that channel, a significant accuracy boost is achieved.

With the combination of methods here and with 4-bit (average) integer quantisaiton, they amazingly achieved only 1% accuracy loss compared to the non-quantised base model VGGNet [SZ15].

2.5 Quantisation-Aware Training

Quantisation-aware training is probably the most promising area of quantisation these days. Since quantised values do not lend themselves to gradient-based training, quantisation-aware training approaches will preserve the full-precision 32-bit floating point weights during training.

During forward propagation the weights and inputs are quantised so the output is based on the quantised values. But during backpropagation, the gradients are 32-bit floating points, and are used to update the full-precision weight values. After the training stage, the production model can then use the quantised values.

Hubara et al. (2016) [HCS+16] use quantisation-aware training to create binarised neural networks (BNNs). A BNN sets all weights and activations $\in \{-1, 1\}$. With these values, major resource gains are made in terms of computation and memory. The memory gains are obvious as a 1-bit value is replacing a 32-bit value. In terms of computation, the speedup is massive, as each multiplication can be replaced by an XNOR function, which is built into the hardware, since the -1 value is represented to the hardware as a 0.

They test both deterministic and stochastic quantisation. The deterministic quantisation function is $Q(x) = Sign(x)$. The stochastic quantisation function, $Q(x) = Sign(x)$, with probability, $p = \sigma(x)$, otherwise $Q(x) = -Sign(x)$, where σ is a hard sigmoid.

Weights and activations are quantised during forward propagation. In order to calculate the gradients during backpropagation, they use full-precision gradients to update the full-precision weights, and set the derivative of the quantisation function to 1.

This quantisation-aware method of BNN development results in a network that requires 32× less memory, and 32× fewer memory accesses. Furthermore, the bit-wise XNOR and bit count computations that replace the typical floating point multiply accumulations are over 32× more energy efficient.

Amazingly, Hubara et al. (2016) [HCS+16] , achieved state-of-the-art performance on MNIST [LC10], SVHN [GBI+14] and other benchmark data sets, without even having to account for the efficiency gains.

2.6 Knowledge Distillation

Knowledge distillation is an efficiency technique developed with the goal of *distilling* the knowledge in a large DNN or an ensemble of large DNNs into a smaller, more-efficient network. This larger model or ensemble of models has been trained, as is standard, to minimise the log probabilities of the incorrect answers, and maximise the log probability of the correct one. But in practice the model will always assign some probability value to incorrect answers, even if these probabilities are minute.

These incorrect probability values, although incorrect, still include more useful information than they do noise. For example, for a classifier trained on MNIST [LC10], given an image of a '3', the classifier will likely assign a higher probability that the digit is a '2' than it is a '1'.

Hinton et al. (2015) [HVD15] created this method of knowledge distillation as a way of taking advantage of this information, believing that the goal of training a model is for it to learn to generalise, not learn the training data. Using the output data from the larger model can help achieve this.

The larger model(s) act as a *teacher*, and the smaller model as a *student*. The student is trained on the *soft target* output of the teacher, instead of the *hard target* binary values that is the true objective of the model. These soft targets provide more information to the student for each training item, as well as lower variance in the gradients between

training items. With this, Hinton et al. (2015) [HVD15], were able to train the student model with less data and a higher learning rate.

The team found they had best results when they used a combination of the soft target and hard target, heavily favouring the soft target, to train the student on. As when the small model was unable to match the soft target, failing in the direction of the true objective is preferable.

Hinton et al. (2015) [HVD15] tested knowledge distillation on the MNIST data set [LC10], and halved the error of a small network with two hidden layers of 800 neurons. The team went further and trained a student network on the distilled knowledge of a teacher network, and omitted all '3's from the training set for the student. This student network still managed to correctly classify 98.6% of '3's, without ever training on one.

2.7 Weight Sharing

Weight sharing is another useful technique to reduce a model's size. Chen et al. (2015) [CWT+15] propose a framework called *HashedNets*, to limit the memory cost of a model. They use a hash function to group all of the weights in a network evenly into *buckets*, such that each member of a bucket has the same weight.

Where a typical neural network layer is structured as a weight matrix, here the weights of a layer are stored in a weight vector, $w \in \mathbb{R}^{K^l}$, of a certain predefined size, K^l. The weight matrix is replaced with what they call a *virtual weight matrix*, $V^l_{i,j}$, where each matrix element is assigned to a value in the weight vector according to a hash function, $h^l(i,j)$, that map a key, (i,j), to a value, $\{1,...,K^l\}$.

$$V^l_{i,j} = w^l_{h^l(i,j)}$$

Forward propagation is simple, same as a standard network, except for weights being looked up in the weight vector by hashing the input key. For back propagation we are essentially summing the gradients of each item in a weight bucket to calculate the gradient of that weight.

Chen et al. (2015) [CWT+15] found that HashedNet, with a compression factor of 64, achieved 42% of the error of a standard neural network of equivalent size on MNIST [LC10].

2.8 Neural Architecture Search

Neural architecture search attempts to systemically find the best DNN architecture based on whatever objective we have. By parameterising scalable elements of the neural network architecture, we can search through the parameter space to optimise for our objective. Generally the search process requires training many different neural networks just to test each one's performance, which makes for a very time consuming optimisation process. This is probably the newest technique in the domain of resource efficient neural networks, but is promising in cases where the resources are available at training time.

Tan et al. (2019) [TL19] use neural architecture search to create their framework, *EfficientNet*, with the built-in goal of resource efficiency. Commonly neural architecture search will search through the space of one of three network dimensions:

- Depth, the number of layers, d.
- Width, the size of individual channels in a layer, w.
- Resolution, the number of channels in a layer, r.

They propose that the best use of resource in a neural network comes from scaling these three dimensions simultaneously. This *compound scaling method* will attempt to optimise the balance between four parameters: $\alpha, \beta, \gamma, \phi$. Where $d = \alpha^\phi, w = \beta^\phi, r = \gamma^\phi$.

The first step is to find a good balance between α, β and γ. Since the number of FLOPs of a neural network is proportional to $d \times w^2 \times r^2$, they begin by fixing $\phi = 1$

and $\alpha \times \beta^2 \times \gamma^2 \approx 2$. With this a grid search is performed across the space of α, β and γ.

With α, β and γ optimised, they are constrained to their optimal values, and now ϕ is iteratively scaled up. As each scale up is a reduction in network efficiency, when the network search achieves the target accuracy we can stop scaling up.

In testing on ImageNet [DDS+09], Tan et al. (2019) [TL19] with *EfficientNet* consistently achieve comparable accuracy to competitors that are 10 times its size.

2.9 Retraining From Scratch

This idea from Liu et al. (2019) [LSZ+19] is not a resource efficiency technique per se, but it is still interesting and worth exploring. When we implement a pruning method onto an over-parameterised neural network, the typical *prune then retrain* pattern makes an important assumption that may not be true at all. It assumes that the remaining weight value after pruning are crucial to achieving a resource-efficient model.

Liu et al. (2019) [LSZ+19] test this assumption by tweaking the pruning algorithm, such that after the parameters are pruned, they reinitialise the model's weights and train it from scratch.

Testing on multiple pruning methods, including weight-level pruning [HPTD15] and network slimming [LLS+17] (which is this author's own work from two years previous), the researchers find that the results achieved by retraining from scratch after pruning are very similar to the results from fine-tuning the existing weights.

In this way, network pruning can be reimagined as a form on neural architecture search, where pruning is used to find the smallest possible model that can rival the base model in terms of accuracy.

2.10 Technique Combinations

This is the area of useful knowledge within the realm of research on DNN resource efficiency that is the most incomplete. There is an abundance of research on proposed efficiency techniques, but when it comes to testing combinations of existing techniques, there is some good research, but much to learn.

Han et al. (2016) [HMD16] developed a framework called *Deep Compression* in which he tests a pipeline of three techniques in sequence:

1. Weight-level pruning, in which they use their own work, [HPTD15].
2. Weight sharing, drawing on *HashedNets* [CWT+15].
3. Compression, using Huffman Coding [Fur06].

The first technique reduces the number of paramaters to make a smaller neural network, the weight matrices are pruned, retrained and made into sparse matrix format.

For the weight sharing stage, the weights are clustered into 256 buckets for each layer. Here, instead of using a hash function to randomly initialise the weight sharing, we already have weights to preserve, so the weights are clustered based on value. A weight vector is created with the centroid of each cluster, and each weight in the matrix can be replaced with relevant 8-bit index to the weight vector. The model is further trained to fine tune the weight vectors.

Finally, since some of the clusters are much more populated than others, we can compress the entire network using Huffman Coding. This technique will only give benefit in terms of storage requirements, as loading the model into memory for use would require decompression.

They tested this triple technique pipeline on AlexNet [KSH12] and achieved a 35× size compression without impacting accuracy.

Chen et al. (2015) [CWT+15] tested the combination of weight sharing and knowledge distillation. $HashedNet_{DK}$ is the neural network they produced in which the weights are randomly initialised into buckets, but instead of training on the hard target objective, they train on the soft target output of the base model. This produced significant improvements over *HashedNet*.

Chapter 3

Methodology

3.1 Neural Network Model

In order to do a comparative study of various methods to help optimise a deep neural network in terms of resource usage efficiency, we will need a trained base model neural network. This base model is used as a basis of comparison for the various methods we will be testing, thus the base model is a large scale over-paramaterised DNN such as those typically used for computer vision. Furthermore, most of the methods we are looking at, including pruning, quantisation and knowledge distillation, will require a large base model as a starting point to achieve efficiency from.

Our use-case for this thesis is ship detection. This means filtering out images of plain ocean, which provide no value to the users on the ground, and just transmitting the images that contain ships. For this use-case there are several styles of computer vision based neural networks, including:

- Binary classifiers. These will output a single binary value that represents 'ship' or 'no ship' for the image.
- Pixel-wise classifiers. These will output a binary matrix in the shape of the input image, so each pixel in the image is classified as 'ship' or 'no ship' [GLL+18].

- Bounding box classifiers. These will output a set of values that describe potential bounding boxes around ships in the image [LZLL18].

A bounding box classifier is probably the most useful for filtering satellite imagery. Compared to a binary classifier, the bounding-box approach gives data that will enable image cropping, and thus less data will be required to transmit. Compared to the pixel-wise classifier, the bounding-box approach lends itself to smaller DNNs, which is important to us in this thesis. Figure 3.1 gives an example of how a bounding box classifier is used.

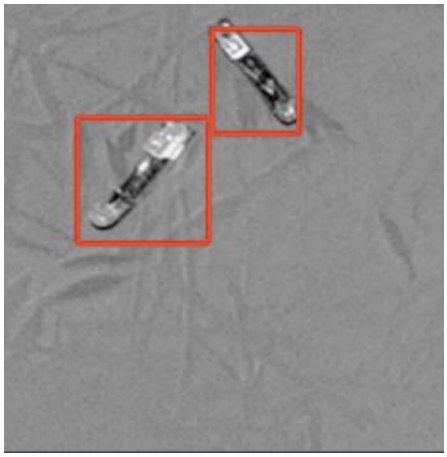

Figure 3.1: Example of bounding box classification [LZLL18].

Under the umbrella of the bounding box classifier, there are several neural network architectures to choose from:

- Deformable parts models [FGMR10], use a sliding window pattern using a classifier network. This means running the neural network many times for each input image.
- R-CNN models [Gir15] use a multi stage region selection and refinement pattern. This involves feature extraction with a CNN and a support vector machine to further quantify these features.
- YOLO (you only look once) architecture models [RDGF16] produce similar or better accuracy in the models with only one neural network inference per image.

We will be implementing a YOLO style architecture for our bounding box classifier. YOLO architecture is state-of-the-art and has achieved great results. The way it works is by dividing the input image into an $S \times S$ grid. Then for each cell in the grid, we predict B bounding boxes centred in that grid cell. We get a confidence value (c), coordinates of the centre position of the box (x, y), and the size of the bounding box (w, h). Figure 3.2 shows a representation of the YOLO architecture.

Type	Filters	Size	Output
			768
Convolutional	32	3x3 / 3	256
Convolutional	64	3x3 / 2	128
Convolutional	128	3x3 / 2	64
Convolutional	64	1x1	64
Convolutional	128	3x3	64
Residual		/ 2	32
Convolutional	256	3x3	32
Convolutional	128	1x1	32
Convolutional	256	3x3	32
Residual		/ 2	16
Convolutional	512	3x3	16
Convolutional	256	1x1	16
Convolutional	512	3x3	16
Residual			16
Convolutional	256	1x1	16
Convolutional	512	3x3	16
Residual		/ 2	8
Convolutional	1024	3x3	8
Convolutional	512	1x1	8
Convolutional	1024	3x3	8
Residual			8
Convolutional	512	1x1	8
Convolutional	1024	3x3	8
Residual			8
Convolutional	10	1x1	8

Figure 3.2: Base Model Description.

Our model most closely resembles YOLOv3 [RF18], although significantly reduced in order to be able to train it on our GPU. We found that YOLOv1, which uses fully connected layers at the end, is too dominated by the size of the fully connected layers, which are unnecessary in object detection as the relevant information to an object is

local, not global in the image. So devoting the available resources to convolutional layers is better.

We also use residual blocks [HZRS16] to help avoid issues of vanishing gradients and accuracy saturation. Each residual block consists of a reduction layer with a (1×1) kernel, followed by a typical (3×3) convolution layer. For each convolutional layer we use the typical accompaniments including dropout, batch normalisation and leaky ReLU activation. With 19 layers total, the model is properly described in Figure 3.2.

The model output is $2 \times 5 \times 8 \times 8$, where the image is divided into an 8×8 grid. Each grid cell predicts 2 bounding boxes described by the 5 parameters (c, x, y, w, h).

3.2 Frameworks and Tools

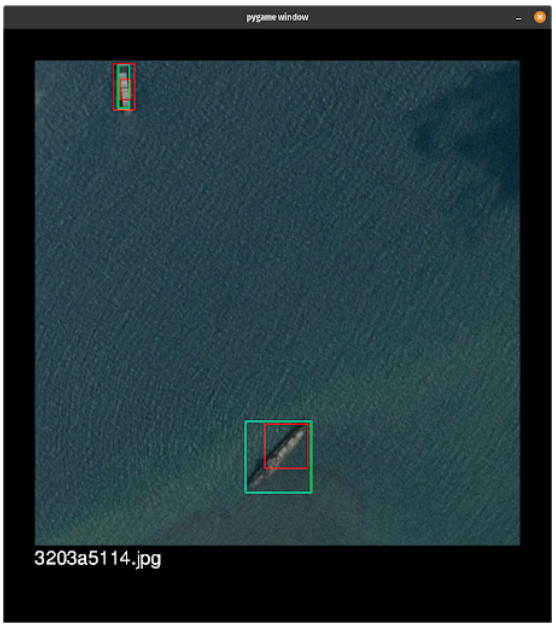

Figure 3.3: Bounding-Box GUI.

All code was written using Python using the typical libraries, with all machine learning work being done using Pytorch. Furthermore, a simple GUI tool was developed using Pygame. This is for debugging purposes, as well as viewing the output of the neural networks in context, i.e. showing the inferred bounding boxes in-place on the image in

question. Figure 3.3 shows the GUI tool, with ground truth in green and model output in red.

3.3 Hardware

For model training and correctness testing we used a NVIDIA RTX3070 GPU with 8GB RAM. Due to the recent chip shortage, GPUs have been in short supply and this was the largest GPU we could reasonably obtain. The size of the models we were able to train, were restricted by the limited memory here.

Figure 3.4: Powertech power analyser setup to in series with Jetson power cable.

In order to test inference timing and power comsumption of our models we used a NVIDIA Jetson AGX Xavier. This computer is made for embedded edge-style systems, and is the kind of machine that would be placed on a nanosatellite. Setting the Jetson to it's 10 Watt setting (it's lowest power setting), provides a good simulation environment for the limited space and power availability conditions on a satellite.

In order to measure the actual power and energy usage of the Jetson during inference we use a Powertech power analyser. This integrates power usage over time. We extend the Jetson power cable by placing the power analyser in series between the Jetson and the power cable's transformer with the help of Anderson power connectors. This allows measurement of the DC current and voltage used by the Jetson. Figure 3.4 shows our power analyser setup.

3.4 Data Set

Figure 3.5: A few images from the data set.

Our selected data set is an open data set from the Airbus Ship Detection Challenge [Air]. This is an ideal data set to use on this project, with 29 GB of image data, that is over 190,000 images. The images are 768×768 pixel RGB JPEGs. Images include

open ocean, shore, ports and land, with about 22% of them including ships. Figure 3.5 shows some of the images in the data set.

Image labels are pixel-wise, given in run-length encoding format. This can simply be transformed into a bounding box format for our uses.

3.5 Training Process

We divide the data set into a training set (75%) and two test sets (12.5% each). We take two test sets to account for the selection bias when choosing techniques to train in combination based on the first test set, ultimately based on our results with individual techniques we chose not to train technique combinations.

We train our model for 125 epochs, with each epoch requiring about 55 minutes to train. We use a batch size of 14, the maximum our GPU could handle with our model and corresponding gradients in memory as well. A typical stochastic gradient descent optimiser is used, with a momentum of 6×10^{-1}, a weight decay of 1×10^{-2}, and a learning rate exponentially decreasing from 2×10^{-3} to 4×10^{-6}. Our data loader is set to shuffle the images, as well as random rotation and flipping. The loss function is based on the YOLO loss function.

YOLO Loss

Our loss function is based on the YOLO loss [RDGF16]. It is divided into three parts, with the loss being a weighted sum of each (weights determined based on tweaking to find ideal numbers):

- The first is the mean squared error (MSE) loss on the confidence values for bounding box predictions with no actual ship in the ground truth. This is given a weight of 12, to account for the overabundance of grid cells with no ship.

- The second is the MSE loss on the confidence values for bounding box predictions with a ship in the ground truth, with a weight value of 1.
- The third is the coordinate loss on the bounding box predictions with a ship in the ground truth. This consists of the MSE on the x and y box position values, and the $sqrt(w)$ and $sqrt(h)$ box size values. This is weighted with a value of 6.

3.6 Efficiency Techniques

For our investigation we have implemented and evaluated five promising techniques outlined in the literature review, with mixed results. The techniques span each of the three broad categories for efficiency techniques in neural networks:

1. Pruning techniques:
    - Weight-level pruning.
    - Network slimming.
2. Quantisation:
    - Post-training quantisation.
    - Quantisation-aware training.
3. Structural efficiency:
    - Knowledge distillation.

The remaining three techniques described in the literature review were not able to be implemented due to time constraints. We picked the most promising and reasonable techniques however. Weight sharing only gave benefits in the storage requirements of the model on disk. Neural architecture search seems like a promising technique, but requires vast training time and computing resources so is left for future work. Finally, retraining from scratch is not really necessary to test as it gives a new perspective on pruning, but not accuracy gains.

In order to test the trade-off between efficiency and accuracy with these techniques, we use our base model, described above, as a starting point from which to apply our techniques. Details of each technique implementation are described below.

3.6.1 Weight-Level Pruning

For weight-level pruning we take an approach just like Han et al. (2015) [HPTD15]. Weight-level pruning aims to reduce the number of weights in the base model by removing the least important ones, as determined by the weight's magnitude.

Using the Pytorch pruning package, we line up all the parameters within the base model into a single array, then we take the absolute values of the weights and sort the array. With this sorted array we can take a threshold value 20% of the way into the array. Then with the threshold value, we set all of the parameters below the threshold to zero.

With this pruned model, we retrain the model for another 20 or so epochs. Once the model is retrained, we can further prune another 20% and retrain. We iterate to pruning values at 20%, 40%, 60%, and 80%.

Results are tabulated in Section 4.2.

3.6.2 Network Slimming

For network slimming we emulate the approach of Liu et al. (2017) [LLS+17]. Network slimming is a technique of removing entire channels from the convolutional layers in the model. The salience of each channel is evaluated globally within the model, similarly to the weight-level pruning, and we use the scaling factors from batch normalisation layers to determine which channels to prune.

With each convolutional layer in the base model already having a trained batch normalisation layer, we first add an L1 regularisation on the batch normalisation scaling factors to the loss function, and retrain the model for a few more epochs. With this we

line up all the scaling factors within the base model into a sorted array to determine the slimming threshold. Then we can remove all of the convolutional channels with scaling factors below our threshold. By removing channels we are able to remove parameters in both the input and output layer of that channel, giving major gains in reduction.

Since we have residual layers in our model, we must align the slimmed channels on either side of the residual block, so that the activations during the addition phase of the residual block remain coordinated. This gives a further reduction on memory use during forward propagation.

As with our previous technique, we prune 20% of the layers from our base model, then retrain for another 20 epochs. We iterate this pattern to pruning values at 20%, 40%, 60%, and 80%.

Results are tabulated in Section 4.3.

3.6.3 Post-Training Quantisation

With quantisation techniques being more mature and common, we apply the approach for post-training quantisation as implemented in the Pytorch quantisation package, which uses 8-bit integer quantisation. Quantisation transforms the 32-bit floating points of a model's weights to 8-bit integers with a linear transformation followed by a rounding to the nearest integer.

For post-training quantisation, we need to determine the zero-point and scale values of the linear transformations for each layer. To do so, we need to optimise these values to minimise the loss when transitioning to 8-bit integers. We calibrate the values by feeding a small sample of data through the trained base model, about 200 batches. During calibration, an observer function creates a histogram of the outputs of each layer so that the scale and zero-point values optimally spread across the range of 8-bit integers values. Once the linear scaling values are determined, we can quantise our base model to produce a model with integer weights that are 4× smaller than the original model.

Results are tabulated in Section 4.5.

3.6.4 Quantisation-Aware Training

Like with post-training quantisation, we use the Pytorch quantisation package to quantise our model into 8-bit integers. Quantisation-aware training is a way of training the model while simulating quantisation, but using floating-point values so that gradients can be used to train with stochastic gradient descent.

During forward propagation, the model simulates quantised values so that the output of inference is what it would be when the model is actually quantised. Then during back propagation the gradients can be calculated with floating points and the weights can be gradually updated. This is trained on the base model for just a few epochs, and the model is then finally quantised to 8-bit values, again giving a model that is 4× smaller.

Results are tabulated in Section 4.5.

3.6.5 Knowledge Distillation

For knowledge distillation we use the approach of Hinton et al. (2015) [HVD15]. Knowledge distillation is a technique of using a large inefficient model as a teacher to teach the smaller student model. The *soft target* output from the teacher model imparts information in it's slight deviation from the ground truth that help the student model learn.

Our base model is used as the teacher model, and our student model is similar to our base model, but with each layer having 25% fewer channels. We run our base model across the whole training set, our targets for the student model is a linear combination of the *soft target* output from the teacher and the *hard target* ground truth. The soft target is weighted to 25% and the hard target at 75%, this way the information from the teacher model output has a strong effect on the learning of the student model. We

save our new targets in a csv file so that we don't need to run both models at once while training the student.

Results are tabulated in Section 4.4.

3.7 Model Evaluation

As our goal in this thesis is to compare the trade-off between correctness and efficiency with our various techniques, we need metrics to evaluate them both.

3.7.1 Correctness

To measure the correctness of an object detection model the standard metric is mean average precision (mAP). mAP is a measure of the area under the precision-recall curve. The precision-recall curve (exampled in Figure 3.6) shows the trade-off between precision and recall of a model with different confidence thresholds. The confidence thresholds used are between 0.5 and 1 with a step value of 0.05, and each point on the curve depicts the precision and recall at a different threshold.

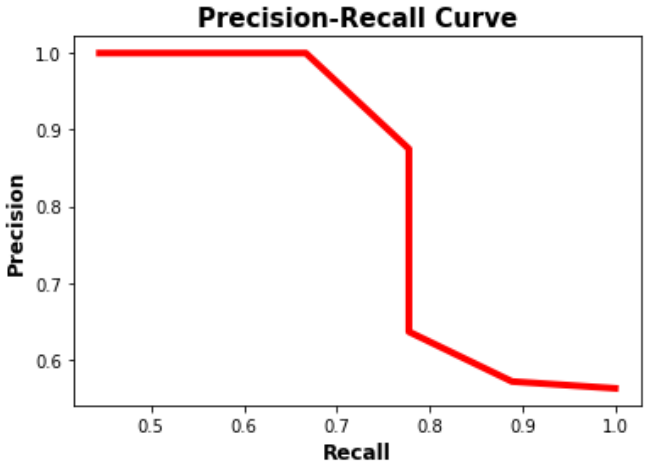

Figure 3.6: Example Precision-Recall Curve [Gad].

In order to calculate precision and recall, where:

$$Precision = \frac{TruePositives}{TruePositives + FalsePositives}$$

$$Recall = \frac{TruePositives}{TruePositives + TrueNegatives}$$

The positive/negative distinction is simply defined by whether the model is confident there is a ship at a specific location, i.e.:

$$Confidence \geq ConfidenceThreshold$$

In order to determine whether a prediction is true or false, we take the intersection over union (IOU) between the predicted bounding box and the ground truth bounding box. This measures the degree of overlap between the prediction and the ground truth (Figure 3.7), i.e. it measure how correct the prediction is. We take an IOU threshold of 0.5.

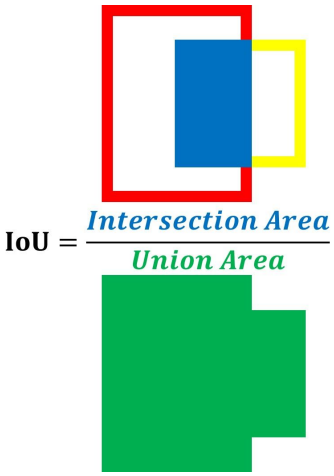

Figure 3.7: Example Intersection Over Union [Gad].

3.7.2 Efficiency

Our efficiency measurements are taken on our NVIDIA Jetson running the neural networks for 250 batches with a batch size of 32. This batch size was chosen as it was found to be the most efficient at running inference compared to other powers of 2. The

efficiency metrics we have collected results for are model size, inference time and energy cost. For each model we take at least 8 measurements to get an accurate mean.

Model Size

This is simply the size of the model in memory or on disk. We will find that the size of the model does not necessarily correlate with inference time or energy cost, which are more relevant to our investigation, as much of the literature seems to presuppose.

Inference Time

When measuring inference time we have a timer collecting the collective time to load the batch onto the GPU and perform an inference with our neural network, as well as the inference time only. The distinction between the two times will be an important part of our findings.

Energy Cost

To calculate the energy cost of running our neural networks we use our power analyser that is connected to our Jetson. The analyser integrates energy usage over time starting from when it is plugged in. So we take an Ampere-hour reading at the beginning and end of the inference run, and the difference between the readings multiplied by the voltage (which is constant at 19.33V) gives us a Watt-hour cost of running the 250 batches. With this we calculate a Joules/image energy cost of each model.

Chapter 4

Results

4.1 Base Model

Figure 4.1 gives our results for the base model, from which we can compare the results of our efficiency techniques. The model we have used as a base model is significantly smaller than a typical production-level computer vision model, and thus has a shorter inference time. Nevertheless, the time to load the image batch and transfer onto the GPU accounts for almost a third of the total time.

Mean Average Precision	0.516
Model Size (MB)	79.6
Average Power (W)	13.1
Energy / Image (J)	1.18
Inference & Load / Image (ms)	90.0
Inference / Image (ms)	60.2

Figure 4.1: Base Model Results.

Due to this effect of significant load time, we decided to collect some figures on the cost of overhead associated with running an object detection model, tabulated in the cost of overhead section. Furthermore, we collect data on a much larger model, in order to compare to the base model, with results tabulated in the larger model section.

4.2 Weight-Level Pruning

Figure 4.2 gives our results for weight-level pruning. We find little loss of accuracy in the 20% and 40% pruned models. However when it comes to the energy cost metrics, we find virtually no difference at all.

	Mean Average Precision	Model Size (MB)	Average Power (W)	Energy / Image (J)	Inference & Load / Image (ms)	Inference / Image (ms)
Base Model	0.516	79.6	13.1	1.18	90.0	60.2
20% Pruned	0.508 (98%)	79.6 (100%)	13.1 (100%)	1.17 (99%)	89.3 (99%)	60.0 (100%)
40% Pruned	0.488 (95%)	79.6 (100%)	13.0 (99%)	1.17 (99%)	90.0 (100%)	60.0 (100%)
60% Pruned	0.432 (84%)	79.6 (100%)	13.0 (99%)	1.16 (98%)	89.2 (99%)	60.1 (100%)
80% Pruned	0.410 (79%)	79.6 (100%)	13.2 (101%)	1.18 (100%)	89.4 (99%)	60.1 (100%)

Figure 4.2: Weight-Level Pruning Results.

Weight-level pruning sets individual weights to zero, but there is no decrease in memory overhead as the weight matrices need are held in dense matrix format in order to execute forward propagation. The storage cost of a weight-level pruned model can be decreased by storing parameters in a sparse matrix format, but this gives no energy gains in terms of execution of the neural network. What our results also suggest is that the energy cost of FLOPs is no less when the input values are zeros.

4.3 Network Slimming

Figure 4.3 gives our results for network slimming. Only the 20% slimmed model gives a workable accuracy loss, beyond that accuracy drops off quickly. We do find significant gains in energy and time efficiency, but not even close to the scale of accuracy drop-off.

The decrease in model size is very good, it decreases with the square of the proportion of remaining channels. Most importantly however, the model size decrease does not cor-

	Mean Average Precision	Model Size (MB)	Average Power (W)	Energy / Image (J)	Inference & Load / Image (ms)	Inference / Image (ms)
Base Model	0.516	79.6	13.1	1.18	90.0	60.2
20% Slimmed	0.488 (95%)	49.7 (62%)	12.9 (98%)	1.14 (97%)	88.3 (98%)	58.5 (97%)
40% Slimmed	0.418 (81%)	27.0 (34%)	12.8 (98%)	1.11 (94%)	86.7 (96%)	57.0 (95%)
60% Slimmed	0.259 (50%)	11.7 (15%)	12.4 (95%)	0.97 (82%)	78.2 (87%)	51.2 (85%)
80% Slimmed	0.115 (22%)	2.7 (3%)	12.1 (92%)	0.91 (77%)	75.2 (84%)	48.9 (81%)

Figure 4.3: Network Slimming Results.

relate well with the energy or time cost. The literature boasts about decrease in model size, Liu et al. (2017) [LLS+17], but when energy efficiency is the evaluation metric, this technique gives negative results. Interestingly, when comparing to our knowledge distillation results we find that smaller models that went through the network slimming algorithm, i.e. channel selection based on batch normalisation scaling factors, perform better than much larger models that are slimmed uniformly.

4.4 Knowledge Distillation

Figure 4.4 gives our results for knowledge distillation. Using a model with 25% fewer channels in each layer, almost half of the total parameters, we train this model twice, once with the distilled knowledge from the base model, and once with the ground truth targets. Similar to network slimming, it is clear that the energy efficiency gains are not on par with the accuracy losses.

Our attempt at knowledge distillation was unsuccessful, the identical model trained on the regular hard targets performed better than our distilled knowledge model. This is likely due to the fact that our teacher model, the base model, had not reached a sufficient level of accuracy across our ship detection knowledge domain. Another contributing factor could be the fact that the size difference, about a factor of 2, is not

	Mean Average Precision	Model Size (MB)	Average Power (W)	Energy / Image (J)	Inference & Load / Image (ms)	Inference / Image (ms)
Base Model	0.516	79.6	13.1	1.18	90.0	60.2
Trained On Distilled Knowledge	0.298 (58%)	44.9 (56%)	12.5 (95%)	1.06 (90%)	84.7 (94%)	54.7 (91%)
Trained On Hard Targets	0.377 (73%)	44.9 (56%)	12.5 (95%)	1.06 (90%)	84.7 (94%)	54.7 (91%)

Figure 4.4: Knowledge Distillation Results.

great enough for knowledge distillation to take effect.

4.5 Quantisation

Figure 4.5 gives our results for quantisation. Here we find strongly positive results amongst a sea of negatives. There is virtually no accuracy loss from quantisation, quantisation-aware training had slightly better results, but we find an energy and time usage gain by more than a factor of 2.

	Mean Average Precision	Model Size (MB)	Average Power (W)	Energy / Image (J)	Inference & Load / Image (ms)	Inference / Image (ms)
Base Model (Float32)	0.516	79.6	307.3	35.4	115.2	114.2
Post-Training Quantisation (Int8)	0.503 (97%)	19.9 (25%)	336.6 (110%)	17.4 (49%)	51.7 (45%)	50.7 (44%)
Quantisation-Aware Training (Int8)	0.524 (102%)	19.9 (25%)	336.6 (110%)	17.4 (49%)	51.7 (45%)	50.7 (44%)

Figure 4.5: Quantisation Results.

Results collection for quantised models had been complicated by a couple factors. Firstly, the NVIDIA GPU back end is not yet compatible with quantised neural net-

works, this is in development however. Thus, we decided to compare float and int models running on a CPU instead. Secondly, we experienced software issues running our quantised model on the Jetson that we connected our power meter to. This is due to some issue in the Jetpack software development kit that are associated with NVIDIA's embedded systems. The issue has been reported, and moderators on the NVIDIA team have replicated the issue, but not yet found a solution. Our solution was to collect energy usage results on a laptop, and use the PyRAPL library to estimate the energy costs.

The drawback of these results is that much of the overhead cost we found in the cost of overhead section do not apply in this case. Background fetching of batches is much more effective here, due to using a much better CPU and not being on a minimal power mode like the Jetson. Also there is not the delay of transferring the data from the CPU's memory onto the GPU's memory. Nonetheless, the results are clear, there is no accuracy loss, with a 2× energy cost decrease and a 4× model size decrease.

4.6 Further Investigations

4.6.1 Cost of Overhead

We have collected results on the energy costs that are not reachable to be made more efficient by our efficiency techniques, these include:

- The costs of keeping the computer on, but not running anything.
- The costs of loading image batches onto the GPU, but not running any model.
- The costs of loading images onto the GPU, and running the batches through the first layer of our base model only.

Figure 4.6 shows what we found.

	Average Power (W)	Energy / Image (J)	Time / Image (ms)
Base Model	13.1	1.18	90.9
Computer Idle	9.6 (74%)	-	-
Image Loading Only	10.9 (83%)	0.39 (33%)	35.8 (40%)
Single Layer Model	11.7 (89%)	0.63 (53%)	53.8 (60%)

Figure 4.6: Base Model Results.

We found that there is a huge impediment to finding major gains in efficiency of running our models. This impediment is that the loading of batches and transferring them onto the GPU takes a third of the energy. Additionally we find that reducing our entire model down to one layer, only allows for a 2× decrease in energy usage. The wattage is not the major factor in reducing energy cost, more so it is the time of execution.

4.6.2 Larger Model

To see how these energy effects scale up to much larger models, we test models with double the layers of the base model. Unfortunately we could not train these models on the hardware available. Figure 4.7 shows that although this model is more than twice as large in terms of parameters, energy and time cost increases by a smaller factor.

	Model Size (MB)	Average Power (W)	Energy / Image (J)	Inference & Load / Image (ms)	Inference / Image (ms)
Base Model	79.6	13.1	1.18	90.0	60.2
Large Model	175.5 (220%)	13.5 (103%)	2.30 (195%)	169.9 (188%)	135.9 (226%)

Figure 4.7: Large Model Results.

4.6.3 Fewer Layers Model

We test out a simple reduced model with 2 fewer residual blocks, that is 4 fewer layers. In order to compare to the more complicated pruning methods of weight-level pruning

and network slimming, it is worth doing a more straight forward model reduction and seeing what we find.

Figure 4.8 shows that the accuracy loss was actually less than the energy cost decrease, unlike our network slimming results. Also the reduction in the actual size of the model was minor, but the decrease in energy cost was significant.

	Mean Average Precision	Model Size (MB)	Average Power (W)	Energy / Image (J)	Inference & Load / Image (ms)	Inference / Image (ms)
Base Model	0.516	79.6	13.1	1.18	90.0	60.2
Reduced Model	0.469 (91%)	74.0 (93%)	12.2 (93%)	1.03 (87%)	84.4 (94%)	55.6 (92%)

Figure 4.8: Fewer Layers Model Results.

Chapter 5

Conclusion

5.1 Discussion

The key achievement of this thesis is the use of the same model and data-set to allow the comparison of many resource efficiency techniques. There is value in directly comparing such techniques this way, as generally the literature on the topic assesses individual techniques. We were able to build good working models, and evaluate for energy cost, using weight-level pruning, network slimming, knowledge distillation, post-training quantisation, and quantisation-aware training. Most of these techniques produced negative results, however there is still much to learn from these investigations.

Our results from weight-level pruning showed no decrease in energy cost at all, indicating that a floating point operation is no cheaper when the input values are zeroes. From network slimming we now know that although decreases in model size is huge, this does not translate to proportionally large gains in energy or time cost for inference. Both of these techniques provide the most benefit in terms of storage cost, not energy cost.

In attempting to implement knowledge distillation, we found that our student model learning on the distilled knowledge produced worse results than our student model

learning on the regular hard targets. What this seems to indicate is that the whole concept of knowledge distillation is ineffective unless the teacher model has reached a certain level of mastery across the learned domain, similar to a real-life teacher.

When it comes to quantisation, we found our only positive results out of all our techniques. We found that quantisation, especially quantisation-aware training, gives very strong gains in efficiency without any accuracy loss at all.

We originally planned to implement and evaluate combinations of these techniques, however due to the negative results with individual techniques, there is virtually no chance that combinations of these techniques would be fruitful.

Figure 5.1 gives us a trade-off ratio between our correctness metric over mean average precision, and our efficiency metric of Joules per image. We find that beyond quantisation, the only model that gives us a better trade-off than the base model is the model in which we removed some layers. This indicates that when the goal is energy efficiency of inference, these more complicated pruning techniques are not necessarily the best approach. On the other hand, quantisation gives major gains in the accuracy-efficiency trade-off.

	Mean Average Precision	Energy / Image (J)	Trade-off Ratio
Base Model	0.516	1.18	1.00
20% Pruned	0.508	1.17	0.99
40% Pruned	0.488	1.17	0.95
60% Pruned	0.432	1.16	0.85
80% Pruned	0.410	1.18	0.79
20% Slimmed	0.488	1.14	0.98
40% Slimmed	0.418	1.11	0.86
60% Slimmed	0.259	0.97	0.61
80% Slimmed	0.115	0.91	0.29
Reduced Model	0.469	1.03	1.04
Base Model	0.516	35.4	1.00
Post-Training Quantisation	0.503	17.4	1.99
Quantisation-Aware Training	0.524	17.4	2.07

Figure 5.1: Model Accuracy-Efficiency Comparison.

The other key achievement of this thesis is being able to simulate the computing conditions of a nanosatellite, an embedded hardware system set to low power usage, and actually collect real energy usage data. A simpler, but less accurate way of collecting energy usage data is by using software. However we wired up a power meter, to our Jetson's power cable to get actual energy usage results.

What we have found is that larger models do tend to draw a slightly larger wattage, but wattage is not majorly dependent on model size. Consequently, inference time per image does correlate well with the energy cost per image.

5.2 Over-Parameterisation

As is well established by Roth et al. [RSZ+20] and others, deep neural networks are typically significantly over-parameterised, meaning there is plenty of room for their reduction. Based on our results, we did manage to prune large proportions of parameters in the model, with a small loss of accuracy. However, in comparison to the reduction achieved on other models in the pruning literature, we found a decrease in accuracy with a smaller proportion of parameters pruned.

This indicates that testing many of these techniques on a smaller than typical computer vision model, leaves less room for reduction without loss of correctness. On the other hand, when it came to testing quantisation, we found no loss of accuracy whatsoever. This suggests that even models which have been minimised, may still be able to go further and achieve more efficiency through quantisation, without accuracy loss.

5.3 Speedup

Amdahl's Law [Amd67] is used to find the maximum expected improvement to a process' latency when part of the process is made more efficient. The law is formulated as follows, where p is the proportion of the process being improved, and s is the speedup

of this part of the process:

$$Speedup_{total} = \frac{1}{1 - p + \frac{p}{s}}$$

Amdahl's law is usually used in the context of parallelisation, but this law has a clear application in our case. With 90.0 ms/image and just 60.2 ms of this is consisting of the actual forward propagation of the input image through the neural network, the remainder being the loading of the image, that leaves a maximum speedup of just 3.0 in the case that inference time reduces by an infinite factor.

Loading the data onto the GPU is a major obstacle to reducing the energy cost of running computer vision models while maintaining accuracy levels. This severely limits the viable trade-off between model correctness and model efficiency.

Perhaps finding a way to reduce loading time would be a good way to increase efficiency. But with our Jetson set to it's lowest power rating, and having a weak CPU, we find that usual techniques such as pre-fetching batches with background workers does not help here.

5.4 Limitations and Future Work

Firstly, we still need to collect energy usage results for quantised models on our Jetson. This way we can better compare the quantisation techniques with our other techniques. However, based on the data we do have, it is clear that quantised models produce the best accuracy-efficiency trade-off.

Secondly, it is certainly worth repeating our technique implementations using a much larger base model, that is more in line with a typical production-level computer-vision model. This way there would be more room to reduce the model size without losing accuracy, and knowledge distillation may work as well.

Finally, based on the negative results we found with network slimming, and the comparatively positive results from simply making a model with fewer layers, neural archi-

tecture search seems like a promising area to experiment with. Due to the computing resources and time constraints of this thesis, we were unfortunately unable to test out neural architecture search, but we would have liked to.

Thus, the next stage of investigation here could be a neural architecture search in which the evaluation function is the ratio between the mean average precision and the Joules per image of the model as in Figure 5.1. Once the ideal neural architecture is found, we can then use quantisation-aware training to further increase the efficiency without loss of accuracy.

Bibliography

[Air] Airbus. Airbus ship detection challenge. https://www.kaggle.com/c/airbus-ship-detection.

[Amd67] G M Amdahl. alidity of the single-processor approach to achieving large scale computing capabilities. *Proceedings of AFIPS Conference*, page 483–485, 1967.

[BNHS19] Ron Banner, Yury Nahshan, Elad Hoffer, and Daniel Soudry. Post-training 4-bit quantization of convolution networks for rapid-deployment, 2019.

[CWT+15] Wenlin Chen, James T. Wilson, Stephen Tyree, Kilian Q. Weinberger, and Yixin Chen. Compressing neural networks with the hashing trick, 2015.

[DDS+09] Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. Imagenet: A large-scale hierarchical image database. In *2009 IEEE conference on computer vision and pattern recognition*, pages 248–255. Ieee, 2009.

[DSD+13] Misha Denil, Babak Shakibi, Laurent Dinh, Marc' Aurelio Ranzato, and Nando de Freitas. Predicting parameters in deep learning. In C. J. C. Burges, L. Bottou, M. Welling, Z. Ghahramani, and K. Q. Weinberger, editors, *Advances in Neural Information Processing Systems*, volume 26. Curran Associates, Inc., 2013.

[FGMR10] Pedro F. Felzenszwalb, Ross B. Girshick, David McAllester, and Deva Ramanan. Object detection with discriminatively trained part-based models. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 32(9):1627–1645, 2010.

[Fur06] Borko Furht, editor. *Huffman Coding*, pages 278–280. Springer US, Boston, MA, 2006.

[Gad] Ahmed Fawzy Gad. Evaluating object detection models using mean average precision (map).

[GBI+14] Ian J. Goodfellow, Yaroslav Bulatov, Julian Ibarz, Sacha Arnoud, and Vinay Shet. Multi-digit number recognition from street view imagery using deep convolutional neural networks, 2014.

[Gir15] Ross Girshick. Fast r-cnn, 2015.

[GLL+18] Rui Guo, Jianbo Liu, Na Li, Shibin Liu, Fu Chen, Bo Cheng, Jianbo Duan, Xinpeng Li, and Caihong Ma. Pixel-wise classification method for high resolution remote sensing imagery using deep neural networks. *ISPRS International Journal of Geo-Information*, 7(3), 2018.

[GYC16] Yiwen Guo, Anbang Yao, and Yurong Chen. Dynamic network surgery for efficient dnns, 2016.

[HCS+16] Itay Hubara, Matthieu Courbariaux, Daniel Soudry, Ran El-Yaniv, and Yoshua Bengio. Binarized neural networks. In D. Lee, M. Sugiyama, U. Luxburg, I. Guyon, and R. Garnett, editors, *Advances in Neural Information Processing Systems*, volume 29. Curran Associates, Inc., 2016.

[HMD16] Song Han, Huizi Mao, and William J. Dally. Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding, 2016.

[HPTD15] Song Han, Jeff Pool, John Tran, and William Dally. Learning both weights and connections for efficient neural network. In C. Cortes, N. Lawrence, D. Lee, M. Sugiyama, and R. Garnett, editors, *Advances in Neural Information Processing Systems*, volume 28. Curran Associates, Inc., 2015.

[HS93] Babak Hassibi and David Stork. Second order derivatives for network pruning: Optimal brain surgeon. In S. Hanson, J. Cowan, and C. Giles, editors, *Advances in Neural Information Processing Systems*, volume 5. Morgan-Kaufmann, 1993.

[HVD15] Geoffrey Hinton, Oriol Vinyals, and Jeff Dean. Distilling the knowledge in a neural network, 2015.

[HZRS16] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 770–778, 2016.

[IS15] Sergey Ioffe and Christian Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift, 2015.

[KSH12] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In F. Pereira, C. J. C. Burges, L. Bottou, and K. Q. Weinberger, editors, *Advances in Neural Information Processing Systems*, volume 25. Curran Associates, Inc., 2012.

[LC10] Yann LeCun and Corinna Cortes. MNIST handwritten digit database. 2010.

[LDS89] Yann Lecun, John Denker, and Sara Solla. Optimal brain damage. volume 2, pages 598–605, 01 1989.

[LLS+17] Zhuang Liu, Jianguo Li, Zhiqiang Shen, Gao Huang, Shoumeng Yan, and Changshui Zhang. Learning efficient convolutional networks through network slimming, 2017.

[LSZ+19] Zhuang Liu, Mingjie Sun, Tinghui Zhou, Gao Huang, and Trevor Darrell. Rethinking the value of network pruning, 2019.

[LZLL18] Shuxin Li, Zhilong Zhang, Biao Li, and Chuwei Li. Multiscale rotated bounding box-based deep learning method for detecting ship targets in remote sensing images. *Sensors*, 18(8), 2018.

[RDGF16] Joseph Redmon, Santosh Divvala, Ross Girshick, and Ali Farhadi. You only look once: Unified, real-time object detection, 2016.

[RF18] Joseph Redmon and Ali Farhadi. Yolov3: An incremental improvement, 2018.

[RSZ+20] Wolfgang Roth, Günther Schindler, Matthias Zöhrer, Lukas Pfeifenberger, Robert Peharz, Sebastian Tschiatschek, Holger Fröning, Franz Pernkopf, and Zoubin Ghahramani. Resource-efficient neural networks for embedded systems, 01 2020.

[SCYE17] Vivienne Sze, Yu-Hsin Chen, Tien-Ju Yang, and Joel S. Emer. Efficient processing of deep neural networks: A tutorial and survey. *Proceedings of the IEEE*, 105(12):2295–2329, 2017.

[SZ15] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition, 2015.

[TL19] Mingxing Tan and Quoc Le. EfficientNet: Rethinking model scaling for convolutional neural networks. In Kamalika Chaudhuri and Ruslan Salakhutdinov, editors, *Proceedings of the 36th International Conference on Machine Learning*, volume 97 of *Proceedings of Machine Learning Research*, pages 6105–6114. PMLR, 09–15 Jun 2019.

[TYRW14] Yaniv Taigman, Ming Yang, Marc'Aurelio Ranzato, and Lior Wolf. Deepface: Closing the gap to human-level performance in face verification. In *2014 IEEE Conference on Computer Vision and Pattern Recognition*, pages 1701–1708, 2014.

[WLCS18] Shuang Wu, Guoqi Li, Feng Chen, and Luping Shi. Training and inference with integers in deep neural networks, 2018.