Adapters in Low-Precision Neural Networks

John Su

A Capstone Project for the degree of
Master of Engineering in Intelligent Information

School of Electrical and Information Engineering
University of Sydney
Australia
25th May 2023

Declaration

This is to certify that to the best of my knowledge, the content of this thesis is my own work. This capstone project has not been submitted for any degree or other purposes.

I certify that the intellectual content of this thesis is the product of my own work and that all the assistance received in preparing this thesis and sources have been acknowledged.

John Su

Abstract

On the extreme end of low precision neural networks are binary neural networks (BNN), where weights and activations are quantized to 1 and -1. BNNs have the potential to run deep networks on dramatically reduced hardware. However, BNNs often need special training methods making transfer learning difficult. Furthermore, it is cumbersome to require multiple versions of the same networks for different domains. To alleviate this, adapters are investigated as a solution to this problem.

Adapters inject additional parameters into the network and are a fraction of the size of the base network. During finetuning, the weights in the BNN are kept frozen and the adapters are trained to repurpose the BNN to the new domain. This eliminates the need to have multiple copies of the same deep network. Instead, adapters can be swapped out to repurpose the base network to a new domain. Because of their smaller memory footprint, multiple adapters can be stored for the same cost as a single deep network to cover multiple tasks.

In this project, three points are examined; the effectiveness of adapters compared to standard transfer learning methods, the training time and finally the size of an adapter needed to maintain high accuracy. Both a serial and parallel adapter approach were considered. Convolutional based BNNs are examined with the task being focused on image classification.

Using pruning and grouped convolutions to reduce weights, experiments indicate that adapters that are between 7-12% of the memory size of the base BNN provide a good compromise between accuracy and low memory footprint. Accuracy of BNNs are only a few percentage points off standard transfer learning methods while providing faster training times and flexibility in design approach. This project opens a path to implementing deep networks on low end hardware via BNNs with adapters being used to tailor the network to the specific task.

Acknowledgement

I would like to thank my supervisor Dr. David Boland for the tremendous amount of help and guidance during this project. Dr. Boland ensured that the project goals were not too ambitious and provided helpful comments of which direction to take the project.

I would like to also thank my Uncle and Aunt who provided housing and support when first arriving in Sydney, allowing me to focus on my studies.

Contents

List of Figures

List of Tables

Glossary

C_{in} Number of input channels. 47

C_{out} Number of output channels. 47

BNN Binary Neural Network. 3, 5, 6, 14–18, 30, 34, 35, 37, 39, 41–44, 46, 47, 50, 52–54, 68, 69

CNN Convolutional Neural Network. 9, 26, 27, 33, 34, 36, 38, 49, 68, 69

LLM Large Language Model. 65, 66

NLP Natural Language Processing. 9, 32, 33, 65

SOTA State of the Art. 65

STE Straight Through Estimator. 6, 7, 34, 42, 43, 50, 75

UDTA Unidirectional Thin Adapter. 8, 9, 39, 46, 53–55, 69, 82

Chapter 1

Introduction

Low precision neural networks have seen increased attention in recent years as a promising method to reduce the computation resources required to run networks. Standard neural networks typically perform computations using 32 bit floating point operations which are relatively expensive and resource heavy compared to their integer arithmetic counterparts. Researchers have demonstrated that networks that are quantized (converted from floating point to integer based arithmetic) can achieve similar results to their floating point counterparts. On the extreme end of this quantisation are neural networks where paramaters and activations are quantised to a single bit representing 1 or -1 [1]. These so called binary neural networks have attracted interest as they have the potential to run deep networks using significantly less power and hardware requirements than standard neural networks while still achieving high accuracy [2].

However, methods to train BNNs tend to be more time consuming and difficult to train than their equivalent floating point 32 network [3, 4, 1]. This also makes the task of transfer learning i.e. retraining the whole model to a new domain e.g. birds species not seen in the original training set equally challenging. Furthermore, fine tuning the entire model for every new domain, is inefficient as an entirely new model must be stored for every different domain. Instead of retraining the entire model, adapter networks, small networks injected into the BNN are trained to learn the domain specific features while keeping the original model parameters fixed. Previously used for natural language processing (NLP), adapters have also been successfully applied to convolutional neural networks and image classification with multiple strategies being proposed [5, 6].

This Capstone project investigates the combination of binary neural networks and adapters as an effective strategy to help overcome shortcomings of binary networks, in particular binary convolutional neural networks, in the task of image classification.

1.1 Motivation

1.1.1 Emerging Trends

Neural networks have seen tremendous growth in recent years in tasks ranging from computer vision and classification to natural language processing. Although the neural network's ability as a universal approximator was known in the early 1990's [7], popularity in neural networks occurred recently in a large part to the exponential increase in computing power and data. This has led to neural networks achieving state of art performances in a wide range of domains ranging from image classification to natural language processing (NLP) to even the games of Chess and Go [8][9]. However, many of state of the art models such as ChatGPT for NLP, require tremendous amounts of computing resources and power with the estimated number parameters being in the hundreds of billions [10, 11, 8]. Even in image classification, models can easily exceed over 1 billion parameters [12] necessitating the use of cloud compute just for inference due to the sheer number of parameters. This can make the usage of neural networks difficult in areas where cloud usage is not feasible or accessible or in tasks where response time is critical such as autonomous vehicles.

On the other hand, the number of edge devices such as phones, internet-of-things (IOT) devices and embedded hardware has also seen a similar rise in usage as the world becomes more connected and electrified [13]. However, the lower computational power and resources edge devices have often limit their ability to leverage state of the art neural networks.

This discrepancy between the trend of neural networks becoming larger and requiring larger amounts of computational resources and the increase in edge devices has created a large interest from researcher in the attempt to bridge this gap. To bridge this gap, researchers have focused on both the hardware end as well as the software end (network architecture, compiler optimisations).

1.1.2 Advantages of Binary Operations

BNNs are particularly attractive due to binary operations being both computationally and memory efficient. In terms of memory, binary weights can theortically be 32x more efficient than regular 32-bit floating point [1, 19]. Furthermore, with weights and activation's being 1 or -1, the operation of multiplication reduces to performing an XNOR operation between two binary variables and summation reduces to bitcouning [20]. As neural networks are comprised entirely of multiplication and summation, BNNs can theoretically operate on minimal hardware and memory.

1.1.3 Advantages of Adapters

BNNs are difficult to train as the quantisation operation (in this case the sign function) derivative is zero everywhere except at zero. As such numerous training methods have been

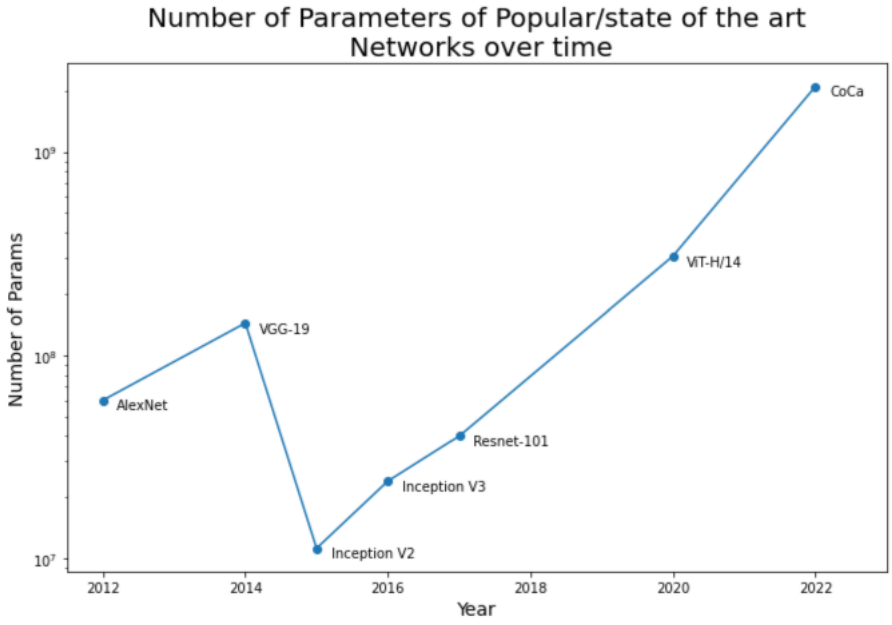

Figure 1.1: Sample of state of the art models in image classification over time in ImageNet classification source [12, 14, 15, 16, 17, 18]

A	B	Output
0	0	1
0	1	0
1	0	0
1	1	1

XNOR Logic

≡

x	y	Output
-1	-1	1
-1	1	-1
1	-1	-1
1	1	1

Binary Multiplication

Figure 1.2: Multiplication when inputs are 1 or -1 reduce to XNOR when a 0 bit represents -1 and 1 bit represents 1. Summation can be done by counting the number of 1 bits - number of 0 bits

proposed [1, 20, 3]. This makes training BNNs often more difficult and slower to train than standard floating point based neural networks. This can make finetuning BNNs

Adapters introduce domain specific parameters for finetuning while keeping the original base weights frozen [21]. This eliminates much of these issues around training BNNs as the adapters can be trained with regular backpropagations (assuming adapters are real valued).

Furthermore, adapters eliminate the need to have an entire neural network for each different domain (e.g. one network for animals, another for cars and another for airplanes). Instead, one can have one network, and have adapters targeted at each of the different domains. This is significantly more memory efficient as adapters are designed to be only a

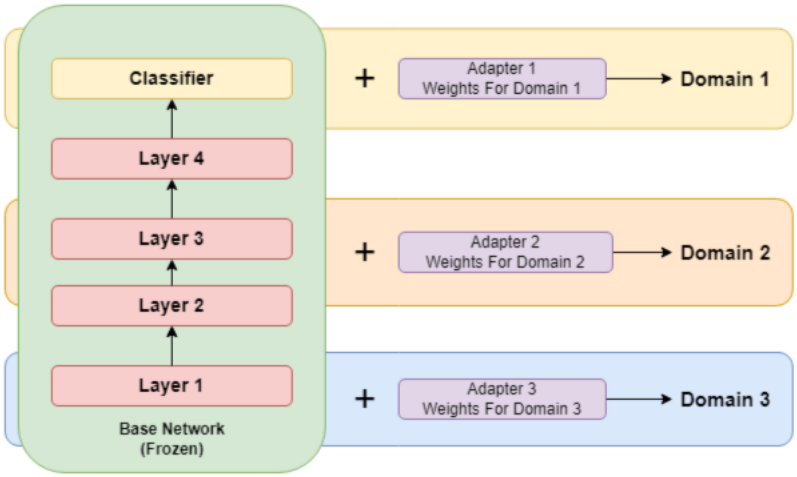

Figure 1.3: Example of how adapters work. To finetune an already trained model, domain specific parameters are added instead of retraining the entire base network again. The base network weights are kept frozen. The number of parameters in an adapter is small compared to the base network allowing for adapters to be easily shared and swapped out [22]

fraction of the size of base network [21, 5]. And because adapters are relatively small, they can be easily shared between users in almost real time [22].

1.2 Research Questions

This capstone project focuses on the software end, focusing on introducing network architecture that can potentially be deployed on low end hardware. The following research question and objectives:

- Can introducing adapters be an effective method to finetune binary neural networks?
- Can one speed up training time by avoiding backpropagation throught the binary neural network?
- How few domain specific parameters can one introduce in the adapter network to achieve high transfer learning accuracy?

These questions are interesting as combining both BNNs and adapters could provide a path for the deployment of neural networks on low powered devices such as phones and embedded devices. The BNN reduces the computational resources while adapters can quickly fit the neural network to the targeted purpose.

Chapter 2

Statement Of Achievements

During this capstone project, the following achievements were made:

- Using adapter finetuning, results in image classification were within a few percentage points off the accuracy of regular finetuning methods while only introducing 8-12% of weights relative to the memory size of the base network. This 8-12% represents a good compromise between accuracy and weight efficiency. As the base network is binary, these adapters are on the order of only a few kB.
- Adapters benefit from increased training speed when used as a method of finetuning vs standard methods. Adapter training can be upto 1.4x faster per epoch, and also reach higher accuracy faster in the initial few epochs than standard methods
- Creation of custom benchmark from Cifar10 and Cifar100 datasets to allow rapid prototyping. These two datasets were each split into 2 subsets; one for training the initial network and the second subset to use as the benchmark for transfer learning.
- Consideration that 32-bit floating point weights are 32x larger than binary weights. This means adapters in this project were made significantly smaller and more compact to ensure the total adapter memory is also a fraction of the base BNN memory.
- Grouped convolution and pruning are both effective methods for reducing the weights in adapters. Grouped convolution is easier to implement but has a limit as to the number of weights to implement. Pruning does not have this limit, but requires an initial overparametization to maintain high accuracy after pruning.
- Based on the benchmarks used, the accuracy of adapters is not strongly dependent on the adapter strategy. The most important factor in accuracy is the number of weights introduced by the adapter.
- Analysis of the strengths and limitations of serial and parallel adapter strategies. Serial approaches are straight forward and quick to implement however require backpropoagtion throught the entire base network. Parallel approaches can completely bypass

backpropagation through the base network but requires modification to the underlying network code to implement.

- Use of the following datasets to demonstrate the effectiveness of adapter finetuning:
  - Cifar10 and Cifar100 [23]
  - Flower102 Dataset [24]
  - Oxford Pets Dataset [25]
- Development of Python code using Pytorch 1.13 [26] that would allow future researchers to implement adapter finetuning for different image classification tasks. For data management, Weights and Biases platform is used [27]. The code can be found here: `https://github.com/Johnnny-suu/Adapters_BNNs`

Chapter 3

Background

3.1 Image Classification

Image classification is a subdomain of computer vision aimed at identifying objects in images. Given an input image with some class e.g. a dog, the objective is correctly identify or 'label' the image as containing a dog. Prior to 2012, non neural networks were used such as support vector machines (SVM) combined with hand crafted feature extraction. Top error rates for the ImageNet Large Scale Visual Recognition Challenge (ILSVRC) hovered around 25% with these methods [28]. In 2012, AlexNet achieved an error rate of 16% for the ImageNet dataset using neural network networks instead of traditional machine learning methods[18]. Since then, neural networks continued to beat out traditional methods in image classification. As of 2022, neural networks now dominate the image classification tasks with current methods achieving 90% in the ImageNet dataset [14, 12].

3.1.1 Cifar-10 and Cifar-100 Dataset

The Cifar-10 dataset contains 60,000 (50,000 training images and 10,000 testing images) 32x32 coloured images each labeled with one of 10 classes. Cifar-10 is a labeled subset of the 80 million tiny images dataset [23]. The Cifar-10 dataset has become a popular benchmark for image classification as the small image size and large amount of labeled data (5,000 images per class) allows researchers to quickly prototype and test different network architectures. It is currently one of the most used image classification benchmarks with over 9000 citations. [12].

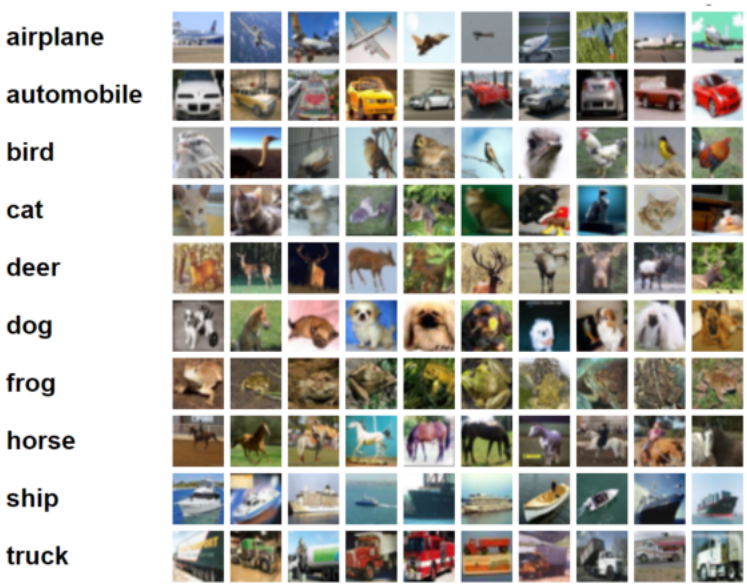

Figure 3.1: Sample of 32x32 images of each class in the Cifar-10 Dataset. Each class contains 5000 training images and 1000 test images [23]

The Cifar-100 dataset also has 60,000 images with the same training and test split but has 100 classes each with 500 training images and 100 test images and thus is significantly more difficult dataset compared to Cifar-10 [23]. The increased number of classes and fewer training images per class make the dataset significantly more challenging than Cifar10.

3.2 Properties of Neural Networks

3.2.1 Neural Networks are Universal Approximators

One of the properties that have made neural networks popular in a wide range of domains is it ability as a universal approximator. The universal approximator theorem demonstrates that neural networks can approximate any function to any level of accuracy. This was first shown in 1989 by George Cybunko who showed that a single hidden layer neural network of arbitrary width with a sigmoid activation function, could approximate any function [7]. Further work by Pinkus in 1999 showed that neural networks with any non-polynomial activation functions, such as ReLu, are also universal approximators [29]. While neural networks have the ability to approximate any function, the theorem does not provide any information on the construction of said neural network nor whether a specific neural network architecture will converge.

3.2.2 Training Neural Networks

Training a neural network is the task of finding an appropriate set of parameters in the network that minimises some objective function given a set of training data. The objective function

acts as a measure of how well the network is performing the task such as cross-entropy for image classification. For most tasks, supervised training is used whereby the training data is pre-labeled with the correct answer. Three stages of training: the feed-forward stage where data is passed through the network, the calculation of some objective function and finally back-propagation of the objective function. The back-propagation algorithm adjusts the parameters of the model slightly to minimise the objective function. This process is repeated to iteratively fit the network to the training data [30, 31].

3.2.3 Feed-forward Algorithm

Neural networks were initially inspired by biological neural networks and are often referred to as *artificial neural networks* [32]. The basic building block of a neural network is the neuron. The neuron take in a weighted sum of inputs and a bias then passes the sum through some non-linear activation function such as the sigmoid or ReLu function. Mathematically for a vector of inputs $\mathbf{x} = [x_1, x_2, ..., x_n]$ the output $f(\mathbf{x})$:

$$f(\mathbf{x}) = \sigma(\sum_{i=1}^{n} w_i x_i + b) \quad , for\ x_i,\ w_i,\ b \in \mathbb{R}$$

where σ is the activation function such as sigmoid or ReLu, w_i is the weighting for input x_i and b is the bias [30]. For multiple neurons in the same layer, calculating each output activation can be represented as matrix multiplication. Mathematically, with n input variables and m output activations:

$$f(\mathbf{x}) = \sigma(\mathbf{W}\mathbf{x} + \overrightarrow{b}) \quad , for\ \mathbf{x} \in \mathbb{R}^n,\ \overrightarrow{b} \in \mathbb{R}^m, \mathbf{W} \in \mathbb{R}^m \times \mathbb{R}^n$$

Where $\mathbf{W}$ is the weight matrix where each row corresponds to the weights of each input for an individual perceptron and $\overrightarrow{b}$ is the bias vector representing the bias of each perceptron. The matrix multiplication produces a output vector of size m. The activation function is then applied element wise [30]. This design is highly flexible and a networks width and depth can be varied throughout the network. The final layer of the network corresponds to the task the network is trying to fit to [30]. In the context of image classification, the final layer is often a fully connected linear layer with a number of output neurons equal to the number of classes to predict. The output neuron with the largest activation is considered the classification of the image by the network.

Matrix Multiplication is highly parallel making GPU's exceptionally well-suited to running neural networks. GPU accelerated training of neural networks were popularized by AlexNet and have what allowed training neural networks to scale to the size that is seen today [18].

3.2.4 Objective Function

The objective function that outputs a scalar value that measures the performance of the network against the training data. The goal of training is to minimise the objective function with the expectation this will improve the network's accuracy. As neural network training uses gradient based methods, objective functions need to be continuous and differentiable and as such methods such as simply calculating accuracy score, which is discrete, cannot be used [30].

For Image classification, cross-entropy loss is often used as the objective function. Given an image x with classification label o:

$$\ell_{cross}(\theta, x, o) = -\sum_{i=1}^{C} y_i \ln(a_i) \tag{3.1}$$

Where C is the number of classes to predict, y_i is a binary indicator variable that is equal to 1 if $i = o$ and 0 otherwise for the image x, and a_i is the activation output for class i [33].

Typically the objective function is averaged over a batch of data containing N data points or images [30]:

$$\ell_{obj} = -\frac{1}{N}\sum_{x}^{N}\sum_{i=1}^{C} y_i \ln(a_i) \tag{3.2}$$

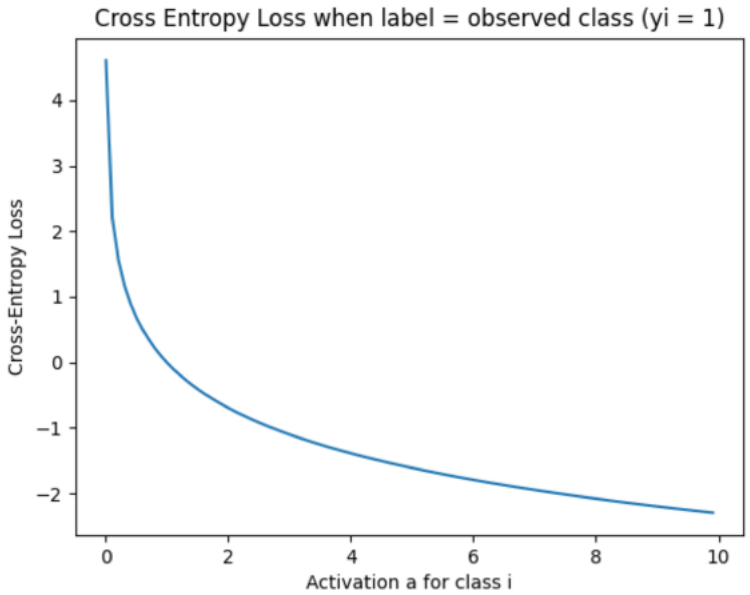

Figure 3.2: Cross Entropy Loss graph when class i is the correct class for image x. As can be seen, maximising a minimises the loss

3.2.5 Backpropagation Algorithm

First introduced in 1986, the backpropagation algorithm provides the foundation for training neural networks [34]. Once the objective function for a batch of data has been calculated, the parameters of the network, θ_t, can be updated using a gradient descent policy [34, 35]:

$$\theta_{t+1} = \theta_t - \frac{\mu}{N}\sum_x \frac{\partial}{\partial \theta_i}\ell_{obj}(\theta_t, x, o) \tag{3.3}$$

Where μ is the learning rate, N is the number of images in the dataset or batch. However, most datasets are large and so updating weights every batch is often too inefficient. However updating the parameters at every training data can produce noisy gradients and the network may struggle to converge [30]. As such, most optimisers today employ the well-know *stochastic gradient descent* (SGD) as a middle ground between the two [31]. Instead of updating every batch or image, SGD updates the parameters over a small subset of the dataset called a *mini batch* of size m [31].

$$\theta_{t+1} = \theta_t - \frac{\mu}{m}\sum_x \frac{\partial}{\partial \theta_t}\ell_{obj}(\theta_t, x, o) \tag{3.4}$$

Typically batch sizes of powers of 2 such as 32 and 64 are used to take advantage of hardware architecture [36]. The training data is shuffled before every epoch to produce different gradients. To calculate the partial derivatives of network parameters neural networks use automatic differentiation. Automatic differentiation utilizes the chain rule and takes advantage of how the derivatives of function's in a neural network are well defined [34].

3.3 Neural Network Architectures

3.3.1 Activation Functions

3.3.1.1 Sigmoid Function

The sigmoid function is a bounded function that maps the $\mathbb{R} \to [0,1]$. Mathematically the sigmoid function and derivative can be expressed as:

$$\sigma(x) = \frac{1}{1+e^{-x}} \qquad for\, x \in \mathbb{R} \tag{3.5}$$

$$\frac{d}{dx}\sigma(x) = \sigma(x)(1-\sigma(x)) \qquad for\, x \in \mathbb{R} \tag{3.6}$$

The sigmoid function can be thought of as a softened and continuous version of the step function. Sigmoid functions suffers from the vanishing gradient problem as gradients for inputs far from the origin saturate to zero. This can make deep networks difficult to train especially for earlier layers [37].

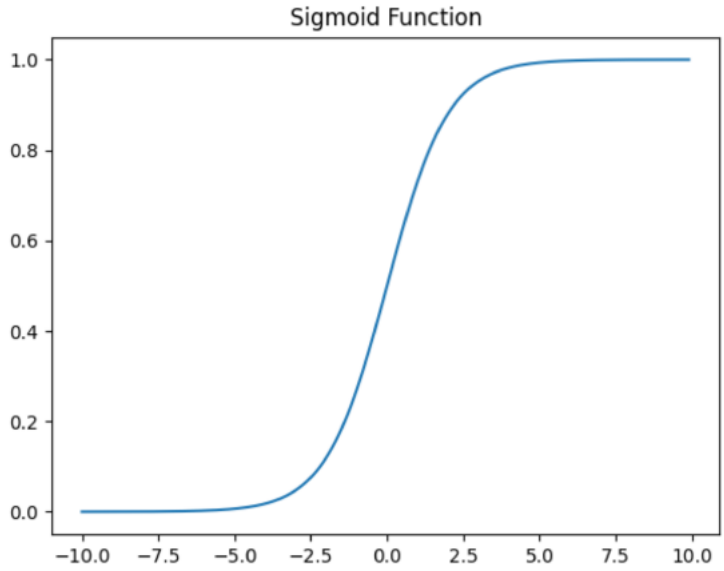

Figure 3.3: Sigmoid Function maps the Real Line to [0,1]

3.3.1.2 ReLu Function

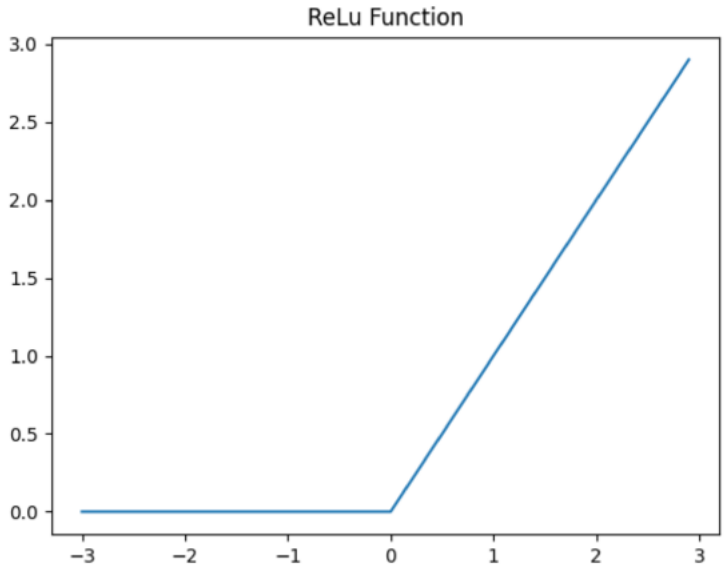

Figure 3.4: ReLu function. The function can also be expressed as $\max(x, 0)$

The rectified linear unit, or ReLu, can be seen in Figure 3.4 and is expressed as:

$$\textbf{ReLu}(x) = \begin{cases} x, & \text{if } x \geq 1 \\ 0, & \text{otherwise} \end{cases} \tag{3.7}$$

$$\frac{d}{dx}(\textbf{ReLu}(x)) = \begin{cases} 1, & \text{if } x \geq 1 \\ 0, & \text{otherwise} \end{cases} \tag{3.8}$$

ReLu function has become popular as it avoids the saturation of gradients that bounded functions such as the sigmoid face and is computationally efficient to compute as the function and its derivative can be implemented as an if statement whereas sigmoid and tanh functions require calculation of the exponential function. However due to the zeroing of negative values and gradient, using ReLu activation functions can result in 'dead' neurons where gradients cannot be propagated through the neuron due to its negative activation [38]. This issue can be some what addressed with batch normalization. [39] Similar to the feed forward algorithm, back-propagation can be implemented in a parallel fashion allowing for accelerated training when using a GPU.

3.3.2 2D Convolutional Neural Networks (CNNs)

2D Covolutional networks have seen significant success in computer vision tasks such as image classification and object detection. Many popular networks such as ResNet,VGG and Yolo implement CNN layers [16, 17, 40]. CNNs were first introduced by LeCun et al. in 1995 [41]. Given a $C_{in} \times W \times H$ Tensor (e.g. an RGB image of size $W \times H$) and output $C_{out} \times W_{out} \times H_{out}$ kernel k, each output channel j can be expressed as:

$$C^j_{out} = b^j_{out} + \sum_{0}^{C_{in}-1} \mathbf{W}_j(k) \star \text{input}(k) \tag{3.9}$$

Where $\star$ is the cross correlation operation. Visually, convolution can be seen as a weight matrix with kernel size k matrix multiplying a patch of the same size as the weight matrix is moved across the grid [42].

The weight matrix can also be thought of as filters that with training, extract useful features from the image such as edges or details such as eyes. Work by Zeiler et al. showed that convolution weights act as filters to extract useful information [43]. Early CNN layers detect basic features such as edges and curves while deeper CNN layers detect more task dependent features such as eyes or head shape [43].

CNNs provide many advantages over regular full connected layers. Firstly, CNN's are computationally cheaper than fully connected layers. For example, for a fully connected layer with an input image of size $|x|$ and to an output size of $|y|$, would require $|x| \times |y|$ connections. This can make even small 3x32x32 images (such as those from cifar10 [23]) infeasible as the number of weights would be $3072|y|$. Conversely, for a convolutions layer the number of weights is independent of the input and only depends on the kernel size (typically 3x3)

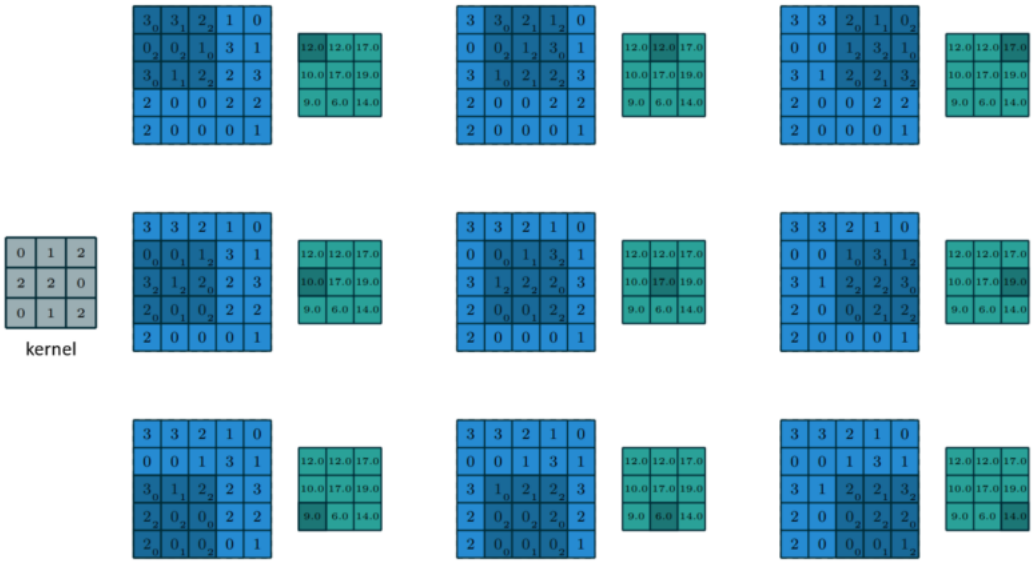

Figure 3.5: Visualisation of a CNN operation. The weight matrix moves across the input tensor performing matrix multiplication with the overlapping input tensor [42]

multiplied by the number of output channels and input channels. Thus, this allows deeper networks to be built with CNN's given the same number of parameters [41, 18].

Secondly, CNNs have built in translational invariance [41]. For computer vision tasks this makes CNN networks powerful as the networks can still recognise object even if it has been moved, rotated or even mirrored. Thus this allows CNN based networks to generalise and train more efficiently than fully connected networks.

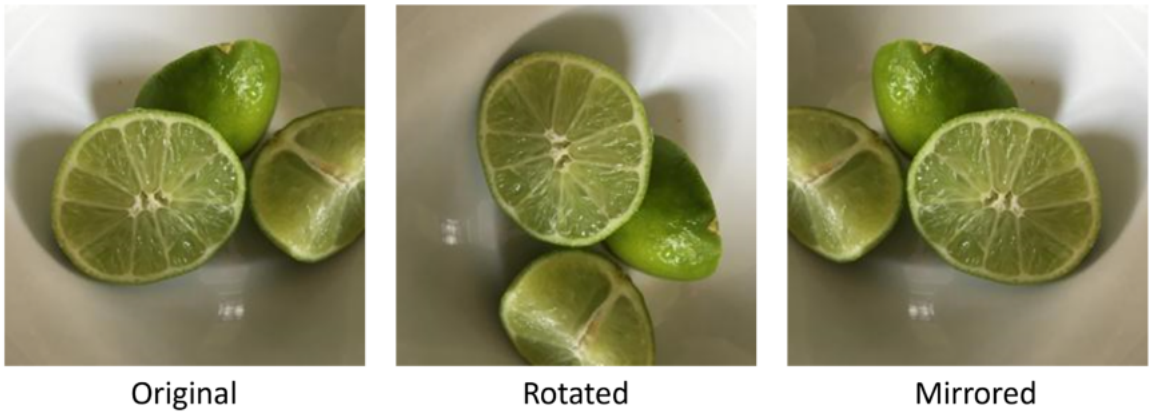

Figure 3.6: Same Image but rotated and mirrored. CNNs have translational invariance meaning they are better able to recognise objects after they have been moved

3.3.3 Batch Normalisation

Batch normalisation was introduced in 2015 by Ioffe and Szegedy [39]. Batch normalisation normalises activation's from a layer to approximately have mean 0 and standard deviation 1. Supposing an batch size of m, activations passing through a batch normalisation layer are scaled and shifted by the mean and standard deviation across the batch:

$$\hat{x} = \frac{x - \mathbb{E}(x)}{\sqrt{Var(x)}} \tag{3.10}$$

Where the expectation and standard deviation are calculated across the batch of training data.

$$\mathbb{E}(x) = \frac{1}{m}\sum_{i}^{m} x, \qquad \sqrt{Var(x)} = \sqrt{\frac{\sum_{i}^{m}(x - \mathbb{E}(x))^2}{m}} \tag{3.11}$$

Finally, the output y is calculated by scaling a shifting the normalised values:

$$y = \gamma\hat{x} + \beta \tag{3.12}$$

where γ and β are learnable parameters. These introduced parameters allow the network to recover the original activation x if the batch normalisation does not help by setting $\gamma = \mathbb{E}(x)$ and $\beta = \sqrt{Var(x)}$. A running average mean and standard deviation is also stored which is then used as the statistics during model evaluations [39].

Batch normalisation help keep activation magnitude near zero and allow for improved training. During training, batch normalisation also act as a regularizer due to the changing statistics across different batches helping to reduce overfitting [39].

3.3.4 Residual Connections

Residual connections were introduced in the seminal paper by He et al.[16]. Figure 3.7 shows how residual connections allows activation's to 'skip' layers allowing information of early layers to propagate to deeper parts of the network. Before residual connections were introduced, deep networks would experience the *degradation problem* where adding layers to a network would actually increase training error [16]. The theory from He et al. is that the non-linear mapping or residual (the layers in seen in Figure 3.7 is easier to learn with the skip connection as worst case the residual can be driven to zero and the identity input can propagate [16].

For an input x, a residual block can be expressed as some non-linear operation $F(x)$ i.e. the layers seen in Figure 3.7:

$$H(x) = F(x) + x \tag{3.13}$$

Note that the output $H(x)$ must be the same shape as the input x. Introducing these skip connections allowed He et al. to successfully train deep networks, and winning ImageNet 2015. Furthermore, skip connections are computationally effiecient as no additional parameters are introduced in the model. Skip connections architectures form the basis of adapters [21].

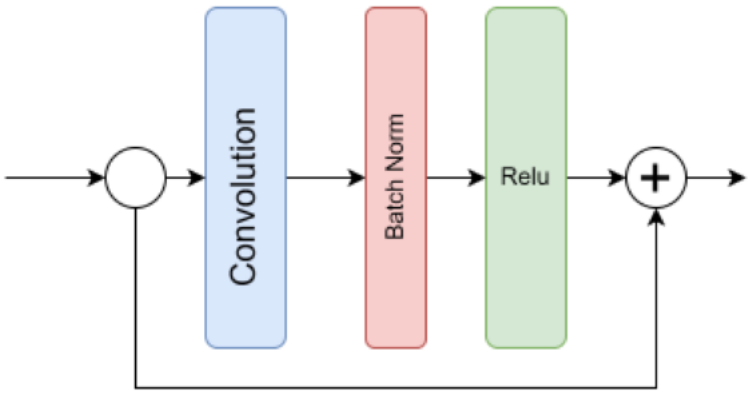

Figure 3.7: Example Diagram of how skip connections work, early activation can 'skip' layers and propagate further into a network. Note that the output shape must match the input shape in a residual network

3.4 Network Optimisation

Neural networks are computationally expensive process. As seen in Figure 1.1 while network performance has been increasing, this has often been accompanied with an increase in computing resources. This makes directly using networks on edge devices such as phones or embedded devices impractical. As a result numerous methods have been presented to make networks efficient.

The most popular method to improve network inference speed is quantising the parameters, activations or both. By default matrix operations are typically performed in floating point 32. Quantisation can increase both inference speed and reduce memory. Quantisation can vary in the extremity. Mixed precision training where single precision or google brain float16 has seen considerable success in improving training time[44]. For use in edge devices, quantisation is mainly focused on integer quantisation where floating point operation are mapped to discrete levels.

3.4.1 Integer Quantisation

8 bit quantisation is currently the most popular form of network quantisation. Pytorch provides a signed 8 bit quantized ResNet-50 model that is is only a few tenths of a percentage points of accuracy off the non-quantized model while being significantly more lightweight and faster [26, 45].

However, quantisation is inherently destructive and non-differentiable [46]. As such, numerous methods are proposed to quantise and train these models [46, 45].

3.4.2 Binary Neural Networks (BNNs)

Binary quantisation is the extreme form of quantisation where all weights are mapped to either 1 or -1. Compared to regular float32 numbers, binary weights can be represented by a single bit. This could result in 32x in memory saving. Using specialised hardware binary arithmetic could theoretically be 32x faster than float32 operations [1]. The speed up is due to multiplication between 1 and -1 is equivalent to an XNOR operation [1] and addition is equivalent to bitcounting [2].

However, with such heavy loss of information binary quantization invariably leads to worse network performance. However the computational savings and inference speed can outweigh this issue. Work by Sun et al. demonstrated a 32x theroitical saving in memory for VGG and Faster R-CNN but only single percetile points loss in classification and object detection respectively [47].

Work by Courbariaux et al used a straight through estimator (STE) apporach to trainingBNNs [1]. Courbariaux et al. achieved comparable results to full precision networks against Cifar-10 and MNIST dataset. The straight through estimator training binarises the activations and weight in the forward pass but passes the gradients to the full precision weight. To improve training, the gradient of the weight r is defined as:

$$g_r = g_q \times \mathbb{1}_{|r|<=1} \tag{3.14}$$

Where g_q is the gradient obtained by backpropagation and $\mathbb{1}_{|r|<=1}$ is the indicator function that is equal 1 if the magnitude of the weight is less than 1 and 0 otherwise. As seen in Figure 3.8, the indicator function is the gradient to a *hardtanh* function and so is equivalent to applying a hardtanh non-linearity directly after an output of a binary layer.

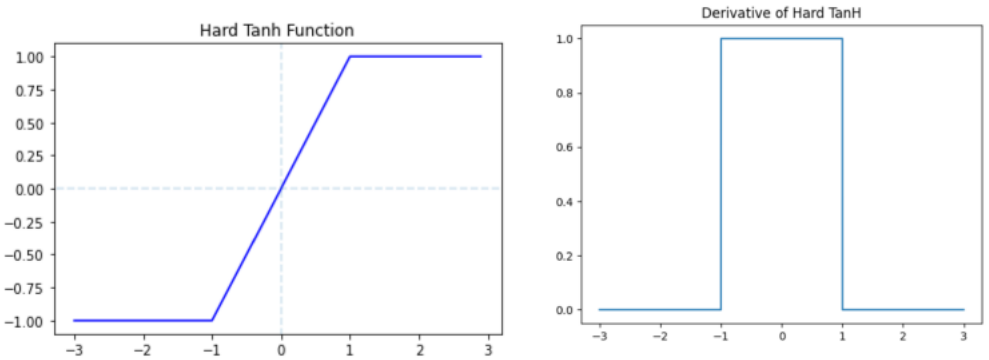

Figure 3.8: Hard tanH function and its derivative

3.4.3 Network Pruning

Pruning in neural networks is an network optimisation technique where weights are removed from the neural network. Researchers have shown that up to 90% of weights can

be pruned/removed without affecting test accuracy [48]. Pruning strategies can range from being completely random to a global strategy where the smallest $p\%$ of weights are removed.

By pruning the network of unnecessary weights, the resulting network can be smaller and be made to faster allowing networks to be better deployed on lower end hardware.

3.4.3.1 Unstructured Pruning

Unstructured pruning strategies ignore the structure of the layers to be pruned. For example, unstructured pruning treats all parameters in a 2D convolutional layer as 'equal' and so each filter would have different number of parameters removed. This pruning technique does not necessarily reduce memory requirements (one must still retain the entire filter) but does make the computation sparser. Sparse tensor computations have the potential to be optimised in the future [48]

3.4.3.2 Structured Pruning

Structured pruning strategies considers the architecture of the layer being pruned. For example, for a 2D convolutional layer, structured pruning can be performed on the by analysing the 'size' of each filter i.e. the smallest filter is pruned. Structured pruning allows for optimisations in both memory and inference as the pruned structure such as a filter can be removed instead of simply setting the weights to zero. The L1 norm is a common criteria to measure the magnitude of a filter:

$$||A||_1 = \sum_{i}^{n} \sum_{j}^{m} |a_{ij}| \tag{3.15}$$

3.4.3.3 Lottery Ticket Hypothesis

Pruning a netowrk while still retaining high accuracy suggests that the initial network is over parameterised. However, if one starts instead with an untrained network with equivalent size and structure to the pruned network often leads to difficulty in convergence and training [48]. As such, one often finds it is better to train the initial over-parameterised network and then prune the network to the desired size. The conjecture behind this is known as the Lottery Ticket Hypothesis which suggests the success of pruning is due to the initial over-parametized network having a higher chance of initialising a smaller sub networks that when trained by itself, can reach equal accuracy to the full network.

3.5 Transfer Learning

Transfer learning is apply a pretrained network to a new domain task. An example of transfer learning is using a model such as Resnet-18 to classify images that the model was not initially trained on such as bird species. Using a pretrained network allows one to leverage the existing

'knowledge' from previous training and allows for significantly faster training time with fewer data points needed to train the network [49].

The standard implementation of transfer learning is finetuning a pretrained network. First the classifier or head of a network with the classifier for the new domain. Then the network weights are retrained to better fit the new domain data[50, 51].

While finetuning the whole has shown to be an effective method of transfer learning, the method has drawbacks that can limit its use. As fine tuning retrains the entire network, for each new task presented, the entire network must be retrained. This can lead to the phenomenon of the network 'forgetting' previous tasks [21] [5]. As such, to avoid model's forgetting what they have learnt, a new model would be needed for each new task. This can quickly become impractical for edge devices where memory is significantly limited compared to data centres.

3.5.1 Adaptive Layers

Rather than train the entire model, Houlsby et al. has suggested injecting the network with adapter layers and then fine-tuning the network using only these layers with the original weights frozen. Houlsby et al. first demonstrated this in the domain of NLP. By using an autoencoder like structure with a skip connection type adapter, Houlsby et al. showed that BERT NLP model could be finetuned to downstream tasks with only 0.4% decrease in accuracy compared to full finetuning [21]. The advantage of using adapters as opposed to

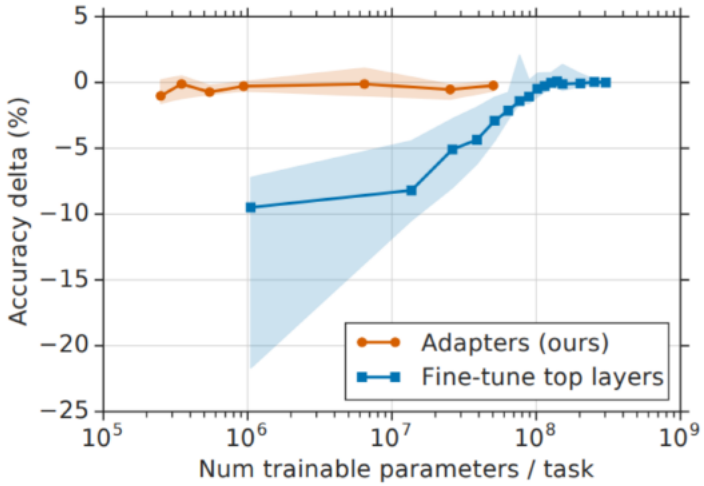

Figure 3.9: Comparison between accuracy for fine tuning using adapter versus standard fine tuning in an NLP task [21]

fully fine tuning is that only a small proportion of weights compared to the full model are needed to fine tune the model allowing for significantly faster training times [21]. Secondly, adapters which are orders of magnitude smaller than the base model can easily be replaced depending on the task, avoiding the need to have an entirely new model for each subsequent

task [21]. The sharability has led to the formation of Adapter Hub where by researchers can easily share and use adapters for different NLP tasks for transformers [22].

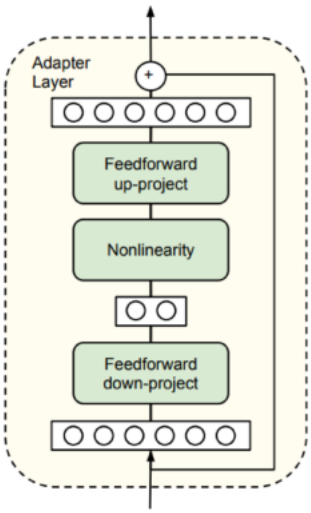

Figure 3.10: Autoencoder design of adapters used in NLP [21].

The success of adapters in NLP and transformer architectures has led to a similar idea propagate to computer vision and CNNs. Rebuffi et al. used a similar resnet CNN like architecture seen in Figure 3.7 where the adapters use a 1x1 convolutional layer [5]. Using these adapters Rebuffi et al. were able to achieve comparable performance against fully fine tuned CNNs models on different image tasks [5].

Chapter 4

Literature Review

4.1 Binary Network Architecture

4.1.1 Straight Through Estimator (STE) BNN

The 'standard' or base BNN used in this project is provided by [1]. The BNN used is a CNN resnet like network as seen in figure 5.1. To train the BNN, the state-through-estimator (STE) is used. During the forward pass, the weights and activations are binarized to 1 or -1 using the sign function. However, during backpropagation, the real valued weights are updated as seen in figure 4.2. This is equivalent to applying an element-wise function G to the weights during the forward pass (in this case the sign function) and treating the derivative of this function equal to 1 everywhere [52].

$$W_b = G(W_r) = Sign(W_r), \quad where \quad \frac{dG}{dW_r} = 1 \tag{4.1}$$

To ensure stable gradient Courbariaux et al also multiplied the gradient by the indicator function:

$$g_r = g_q \times \mathbb{1}_{|r|\leq 1} \tag{4.2}$$

Where g_q is the gradient obtained by backpropagation and $\mathbb{1}_{|r|\leq 1}$ is the indicator function that is equal 1 if the magnitude of the weight is less than 1 and 0 otherwise. As seen in Figure 4.1, the indicator function is the gradient to a *hardtanh* function and so is equivalent to applying a hardtanh non-linearity directly after an output of a binary layer.

4.1.2 ReAct Net

Due to the difficulty and destructive quantisation of BNNs, multiple strategies have been proposed in training BNN [1, 20, 52, 4]. A promising recent method is the ReActNet proposed by Lie et al [4]. Using their methods, Liu et al achieved top 1$ accuracy of 65.4% for their binarised Resnet 18 models without introducing a significant amount of additional parameters unlike the base BNN [4]. For comparison, the regular Resnet18 model achieves a

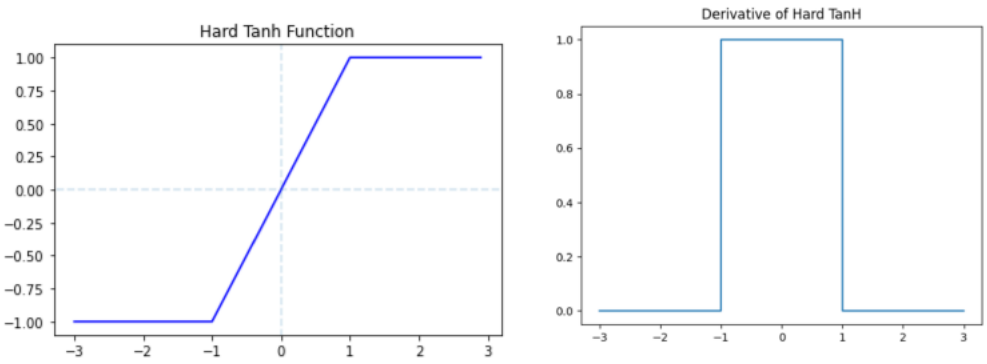

Figure 4.1: Hard tanH function and its derivative

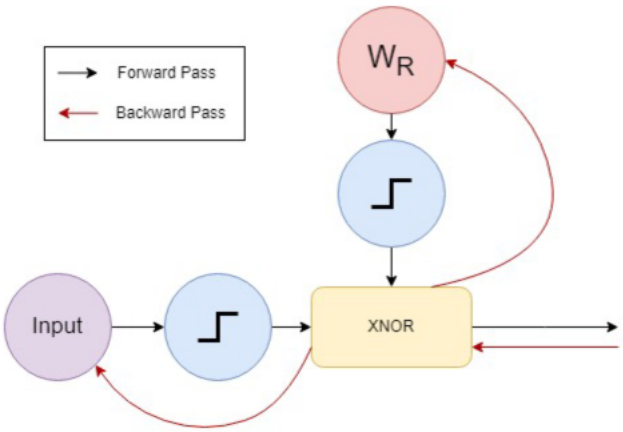

Figure 4.2: Illustration of the state-through-estimator. During the forward pass weights and input activations are binarised but during backpropagation, the gradient updates are passed to the real weights

Table 4.1: Comparison of Top 1% Accuracy For ImageNet on various BNNs based on Resnet18

ResNet18	BNN[1]	XNOR-Net[2]	BiReal Resnet Model[52]	ReAct Resnet Model [4]	Full Precision Model [16]
Top 1% Accuracy	42.2%	51.2%	56.4%	65.8%	69.3%

Top 1% accuracy of 69.8% while other BNN models such as the base BNN and XNOR net achieve only 44% and 51.8% top 1% accuracy respectively [4].

4.1.2.1 Weight Binarisation

$$W_r \approx \alpha W_b, \quad \text{where} \quad \alpha = \frac{||W_r||_{L1}}{n} \tag{4.3}$$

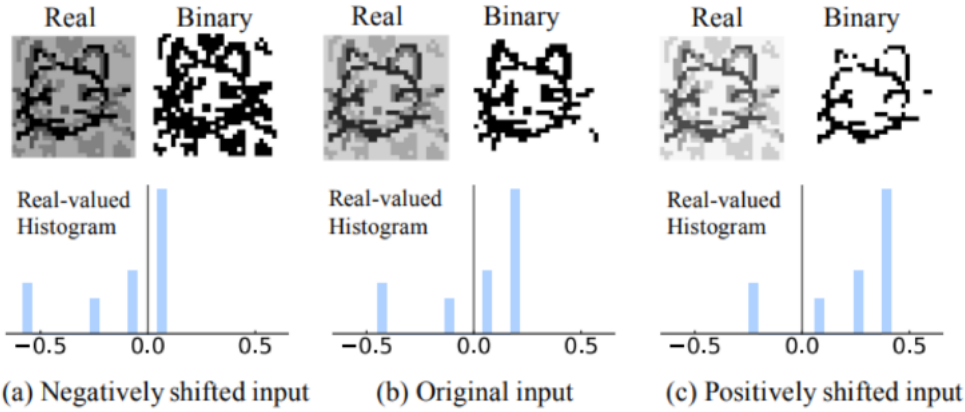

Figure 4.3: Example of by Liu et al on the importance of distribution shift when using binary operations [4]. Using the standard sign function, if the input distirbution of values is too negatively or positviely shifted one either introduces excessive noise or loses important features

Where $||\bullet||_{L1}$ is the L1 norm of the weight matrix and n is the number of elements in the weight matrix. For CNN layers, each filter has its own scalar $alpha$ approximating the original filter.

4.1.2.2 Activation Function

Liu et al argued that the effectiveness of binary based convolutions is highly dependent on distribution of the input feature map [4]. This is because the activation values of binary neural networks are restricted to the values of 1,-1 and so a small shift in input e.g. going from 0.1 to -0.1 can cause a significant change after undergoing binarisation. This can be seen in figure 4.3 where shifting the input feature map negatively or positively shifted input can either introduce excessive noise or cause a loss in features.

To combat this Liu et al modified the binarisation e.g. sign function for activations by introducing a learning threshold parameter for each channel:

$$x_i^b = h(x_i^r; \beta^i) \equiv \text{sign}(x_i^r - \beta_i) = \begin{cases} 1, & \text{if } x_i^r \geq \beta_i \\ -1, & \text{otherwise} \end{cases}$$

Where x_i^r is the real activation, $\beta_i \in \mathbb{R}$ is the learnable parameter and x_i^b is the binary activation of the ith channel. Effectively, RSign adds a learnable bias β that allows the network to shift where the sign function takes place.

The second addition to allow the network to control the distribution is introducing a modification to the PRelu activation called RPreLu. With PReLu, one learns an additional parameter β that controls the slope of the negative values. With RPreLu 2 more learnable parameters:

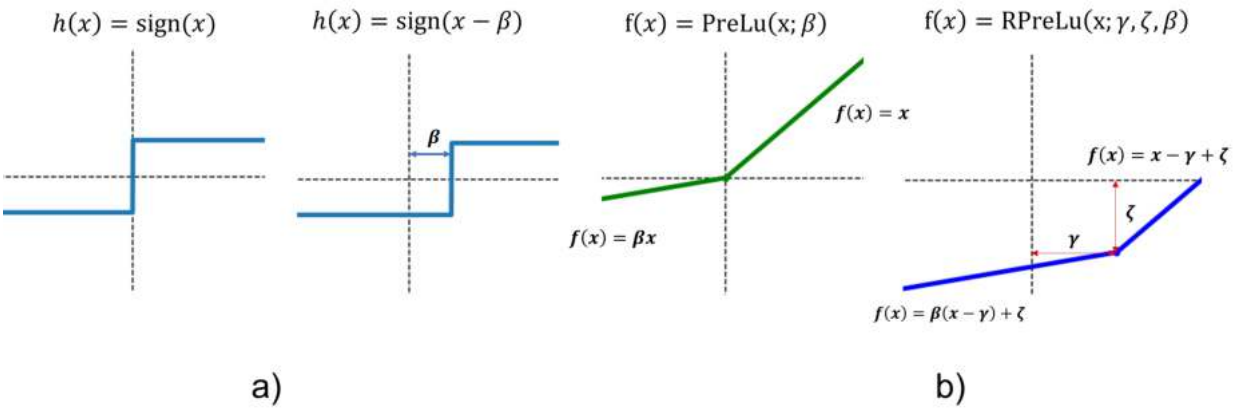

Figure 4.4: Visualisation of the new activation functions introduced for ReAct Net and the effect of each parameter [4]

$$\text{RPReLU}(x_i) = \begin{cases} x_i - \gamma_i + \zeta_i & \text{if } x_i \geq \gamma_i \\ \beta_i(x_i - \gamma_i) + \zeta_i, & \text{otherwise} \end{cases}$$

Here β_i functions the same as in the original PreLU function while γ_i and ζ_i controls the shift of the activation function.

4.1.2.3 Denser Network

A small changed introduced by [52] is to increase the number of skip connections in each residual block. In standard Resnet architecture, each block contains 2 layers of convolutions before being recombined with the original input via a skip connection. However because binary convolution is inherently destructive, [52] applied a skip connection after *every* convolution to limit this loss of information without introducing additional memory overhead.

4.1.2.4 Training Procedure

Finally, BiReAct net uses a 2 step training procedure for training the binary neural network based on the work of [3]. In the first step the network is trained with binary activation but normal convolutional layers containing real valued weights. The initial training serves as a good weight initialisation for fullBNNs [3]. The convolution layers are then converted into the binary convolution layers and the network is now trained as a fullBNN. Using this method [3] were able to train binary networks within 3-5% accuracy point of their fp32 couterpart.

4.2 Adapter Methods For CNNs

Houlsby et al. first demonstrated the use of adapters this in the domain of NLP for large transformer based networks [21]. By using an autoencoder like structure with a skip connection type adapter, Houlsby et al. showed that BERT NLP model could be finetuned to downstream tasks with only 0.4% decrease in accuracy compared to full finetuning [21]. The use of adapters has since been extended to CNN style architectures. Rebuffi et al. was the first to use this approach using a serial adapter approach with resnet like adapter [5]. Since then, other adapter variations for CNN networks have appeared to improve adapter approaches for multitask learning [6, 53, 54].

4.2.1 Serial Adapters

A serial adapter approach injects parameter specific between layers of the existing 'backbone' network. This approach can be implemented with minimal changes to the underlying network code. Rebuffi et al used adapters to achieve comparable transfer learning accuracy compared to standard finetuning where all network parameters are retrained [5]. Using ResNet as the base network, Rebuffi et al's adapters followed a similar resnet like architecture as seen in figure 5.4. To reduce the amount of parameters introduced, Rebuffi used a 1x1 kernel. In this project, the same adapter is used for serial adapters but the final layer is replaced by a hardtanh function (to ensure input values to binary convolutions are between 1 and -1) 5.4.

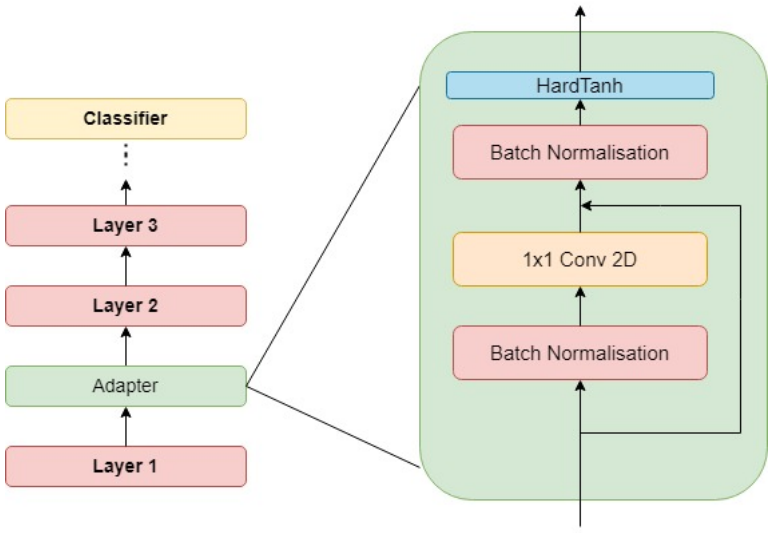

Figure 4.5: Serial Adapter used in this project [5]. The input shape of the Adapter must match the output shape

Because adapters in serial are added between existing layers, the output size and shape must match in input shape [5]. This can limit the design and weight introduction of adapters. For example, inserting an adapter after a layer with C channels, each convolutional layer in

the adapter will introduce $C^2 \times k \times k$ parameters where k is the kernel size. This C^2 factor can be problematic for convolutional networks where the number of channels is high such as the STE BNN discussed in section 4.1.1 where the final layer has 320 output channels. A second issue for serial adapters is that for adapters inserted at earlier layers of a base network, gradients must be backpropagated through a large portion of the base network. For deep networks this can make training slow and for networks with non-standard training methods/architectures such as BNNs training can be more difficult.

4.2.2 Unidirectional (parallel) Adapter

A solution to the above problems is to have the adapter network run completely parallel to the main base network as seen in figure 5.3. Sun et al used this idea creating the so called unidirectional thin adapter (UDTA) [6]. Because the network run parallel to the base network, gradients and backpropagation do not have to pass through the base network to reach adapter parameters. Using this approach, Sun et al reduced backpropagation time by over 80% at small increase in forward pass time compared to finetuning the base network [6]. Furthermore, because the adapter is not placed in between existing layers, one has much more freedom in controlling architecture of the overall adapter networks.

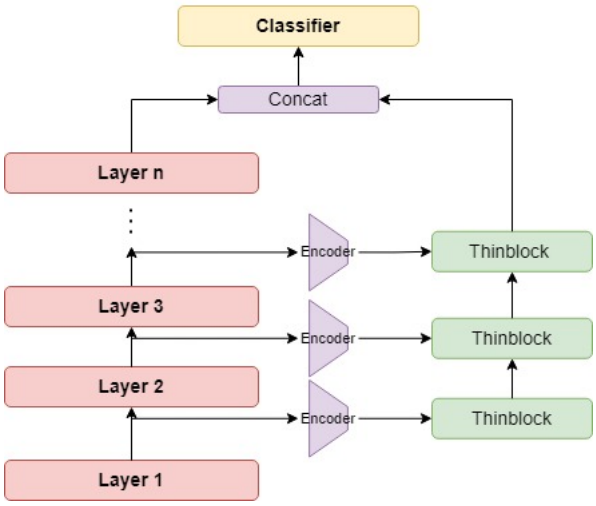

Figure 4.6: UDTA network structure by [6]. Here the encoder and original network (red and purple) are trained on the initial domain agnostic dataset e.g. imageNet and kept frozen. Because the adapter networks are seperated from the original network, backpropagation does not need to go through the original network

Sun et al, used encoders to reduce the feature map size before passing it to the adapter layer [6] as seen in figure 4.6. To limit the amount of additional parameters, this project instead uses average pooling to ensure size consistency between layers. Sun et al first uses a 1x1 convolution layer to increase the number of channels followed by batch normalisation and a ReLu function [6]. Then, Sun et al uses depthwise seperable convolutions in their

so called 'thinblock' adapter. Depthwise, separable convolutions were first intorduced by Mobilenet [55]. Convolution is broken up into 2 steps: 'depthwise' convolution where each channel is convolved independently of one another to capture spatial context (typically a 3x3 kernel). Then a standard 1x1 or 'pointwise' convolution is performed to capture relationship between channels. [55] showed that depthwise seperable convolution can achieve comparable performance while using less operations and parameters.

Chapter 5

Methodology

5.1 Overview

To analyse the effectiveness of adapter networks, several datasets and BNN architectures are used. The Cifar10 dataset [23] is first used as the initial dataset to establish the feasibility of using adapter networks withBNN. The investigation is then scaled to the more difficult Cifar100 dataset [23]. Finally the analysis is scaled up to larger more realistically sized images using the Oxford Pets and Oxford Flowers 102 datasets [25, 24]. The Cifar10 and Cifar100 datasets contain 3x32x32 coloured images while the Oxford Pets and flowers datasets contain much larger RGB coloured images that vary in size.

For Cifar10 and Cifar100, an initial BNN architecture used by [1] is used. The dataset is initially split into 2 subsets; the first subset is used to train the BNN from scratch while the remaining group is used as the finetuning dataset. The BNN for these datasets are first trained to reach comparable accuracy to their fp32 counterparts. For the Oxford Pets and Flowers, a BNN (called BiReal net) that is pretrained on ImageNet is used as the base BNN [4]. The Oxford pets and flowers dataset are then used as the finetuning dataset .

2 adapter strategies are investigated. First, a serial approach is examined whereby adapter networks are inserted in between existing layers [5]. The second approach looks at the so called unidirectional adapter that runs parallel to the base network [6].

Finally, pruning is introduced to reduce adapter network size and analyse the effect on accuracy. During finetuning, the adapter network is first trained for several epochs before being pruned and then further trained in so called 'one shot' pruning method [48].

All training was performed using an 8 GB RTX 2070 Super GPU and AMD Ryzen 3700x CPU using Pytorch 1.13.

5.2 Comparing size of adapters to base network

In previous investigations around adapters, the number of weights could be directly used to compare the size of the adapters with respect to the size of the base network [21, 5, 6].

However,in this investigation the base network is binary while the adapters use 32-bit floating point weights. As such one cannot directly compare number of weights as a measure of size. To better compare the size of adapters, the memory size of each weight (32 for fp32 numbers and 1 for binary weights) is considered. The size of the adapter can then be expressed as a percentage of memory relative to the memory size of the base BNN

$$\text{Size } \% = \frac{32n_{fp32} + n_{bin}}{32N_{fp32} + N_{bin}} \times 100\%$$

where lowercase n denotes the number of floating point or binary weights in the adapter and uppercase N denotes the total number of floating point or binary weights in the base BNN. This metric considers the size of weights and so can more accurately measures the size of the adapter with respect to the base BNN.

5.3 Base Binary Network

5.3.1 STE BNN

This project investigates twoBNNs to be used as the base network for adapter finetuning. The first is the STE BNN by Courbariaux et al as detailed in section 4.1.1 in the Literature Review. [1]. This BNN is first trained from scratch on an initial dataset before undergoing finetuning. This BNN represents a base BNN that many other BNNs are based off [2, 1, 47]. The BNN is applied to the Cifar10 and 100 based benchmarks.

An important feature to note of the BNN used by Courbariaux et al [1] to achieve their results is that the number of weights for the BNN is significantly larger than the equivalent FP32 network it is compared to. If one takes into account the theoretical memory requirements (fp32 parameters requiring 32bits whereas binary weights needing only 1 bit) then the size of each network is comparable. While there may increased amount of multiplications and addition operations with the BNN, this could be offset by the arithmetic operation being reduced to XNOR and bitcounting operations respectively when the BNN is deployed on specialized hardware [2].

Table 5.1: Memory Footprint and number of parameters of BNN by [1] and a full precision Resnet model that they compared with. The memory size is adjusted assuming binary weights are 1 bit and fp32 are 32 bits.

	Binary Weights	fp32 Weights	Total Weights	Memory Size (kB)
BNN	4330165	6250	4336415	566
Full Precison Resnet Model	-	175258	175258	701

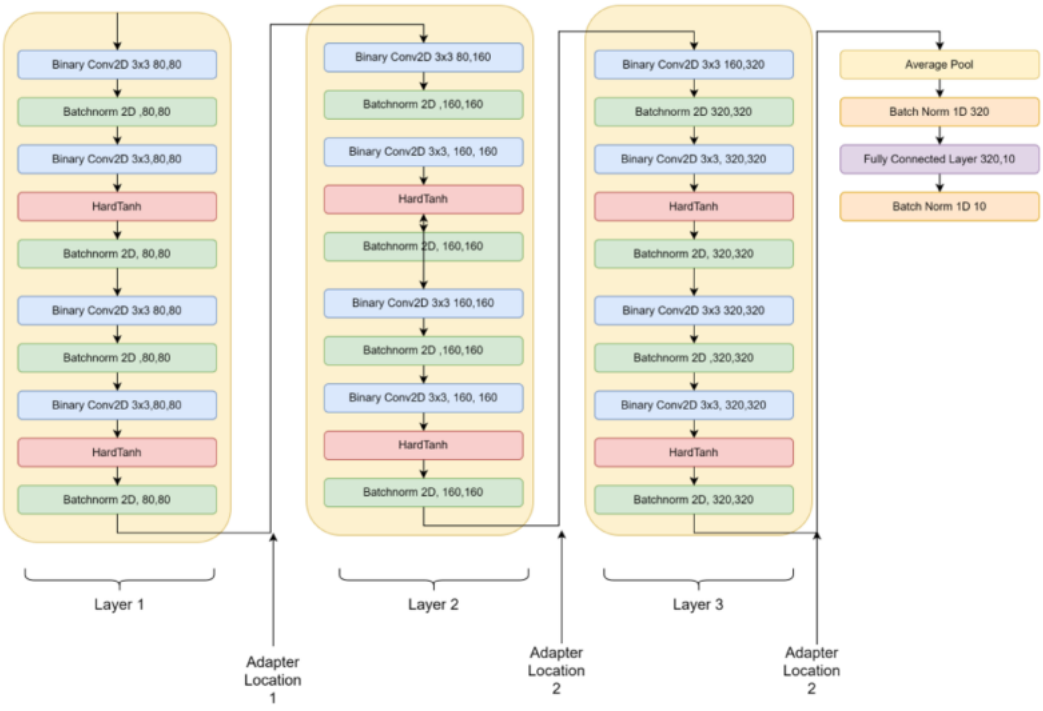

Figure 5.1: Binary Resnet18 Network from [1]. Adapters/feature maps are added/extracted after each layer/residual block

5.3.2 ReactNet

The second BNN model used is a pretrained model on ImageNet from Liu et al, based on a binarised Resnet18 [4] and discussed in section 4.1.2 of the literature review. The BNN has the same number of weights as the regular Resnet18 but most are instead binary rather than fp32. The ReActBNN contains more than twice the number of binary weights that the STE BNN.

Using thisBNN represents a more realistic approach one would take in real-life in taking an off-the-shelf high accuracy pretrained model rather than training a network from scratch first. The high pretrained accuracy also removes any possible errors or results due to poor initial training.

Table 5.2: Comparison of parameters between ReAct Resnet18 Model by [4] and the real valued ResNet18 [16]. When taking into account theoritical memory storage of binary weights, the BNN requires 10 times less memory to store

	Binary Weights	**FP32 Weights**	**Total Weights**	**Memory Size (kB)**
ReAct ResNet18	10,985,472	718,952	11,704,424	4,249
Regular ResNet18		11,689,512	11,689,512	46,758

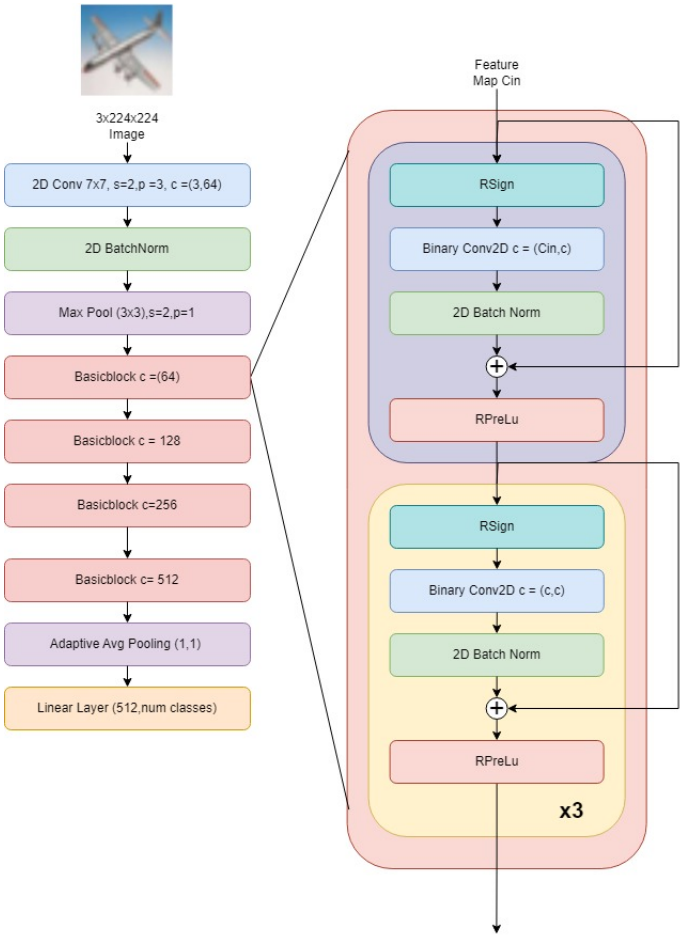

Figure 5.2: Network Architecture of ReAct Net's BNN by [4]. c is the number of channels,p is the padding, s is the stride of the convectional kernel. The architecture is almost identical to that of Resnet18 [16]

5.4 Adapters

For this project, two adapter strategies are explored: Serial and parallel based adapters. Serial adapters are added in between existing layers while a parallel approach extracts feature maps from the existing layers, using them as inputs to the adapters. Both approaches have their strengths and weaknesses. The serial adapter can be implemented with minimal modification to the base network code. However, because the adapter is placed between two existing layers, the input and output shape of the adapter is fixed limiting the design flexibility. Because a parallel approach only uses the base network to extract feature maps, the design and shape of each adapter is more flexibile. However, the additional modifcations to underlying network code must be made. This project investigates two different [**CNN**] based adapter architectures by [5] and [6].

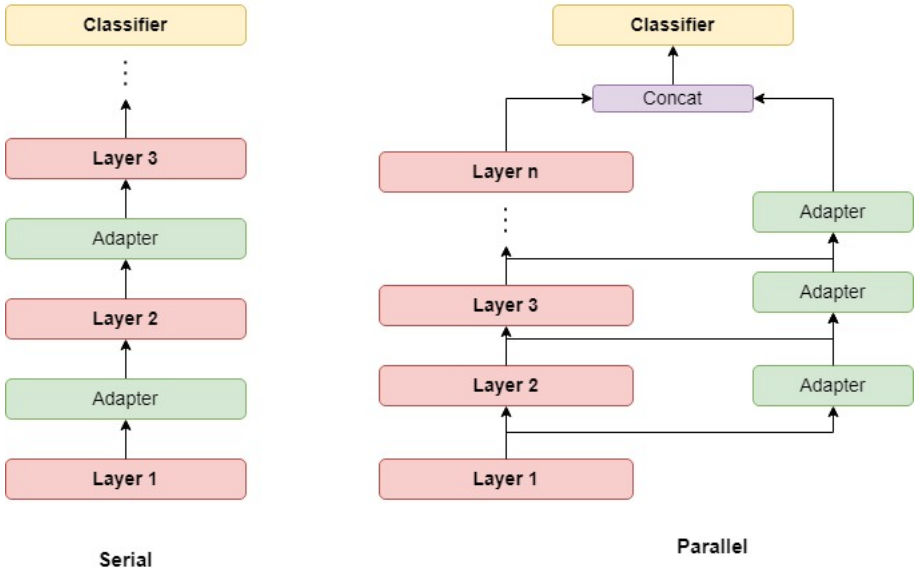

Figure 5.3: The two types of adapter networks investigated in this project.

5.4.1 Serial Adapter Design

For the serial approach, the same adapter architecture from Rebulli et al. is used [5]. However, the final layer is replaced by a hardtanh function (to ensure input values to binary convolutions are between 1 and -1) [1]. Furthermore, the an adapter is placed after each layer similar to figure 5.3.

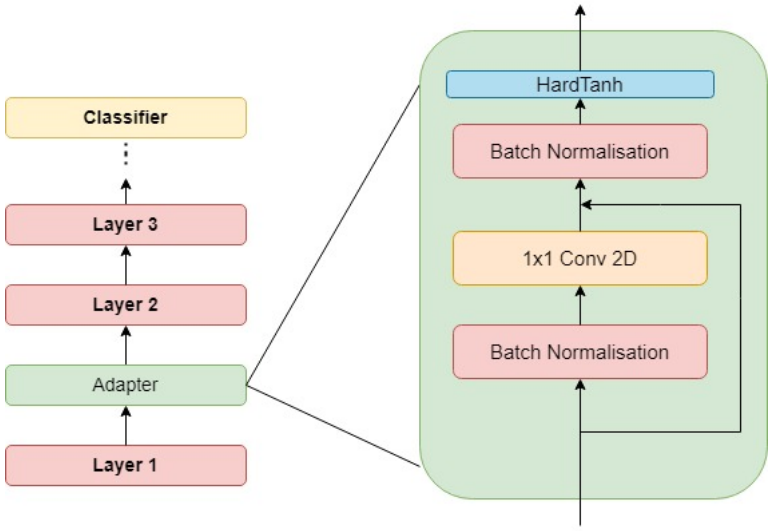

Figure 5.4: Serial Adapter used in this project [5]

5.4.2 Parallel Adapter Design

For the parallel approach, the adapters used is a modification of the UDTA adapter introduced by [6]. In this project, the encoder structure is not included to reduce the number of parameters in the model, being replaced by average pooling to ensure feature map size consistency as seen in figure 5.5.

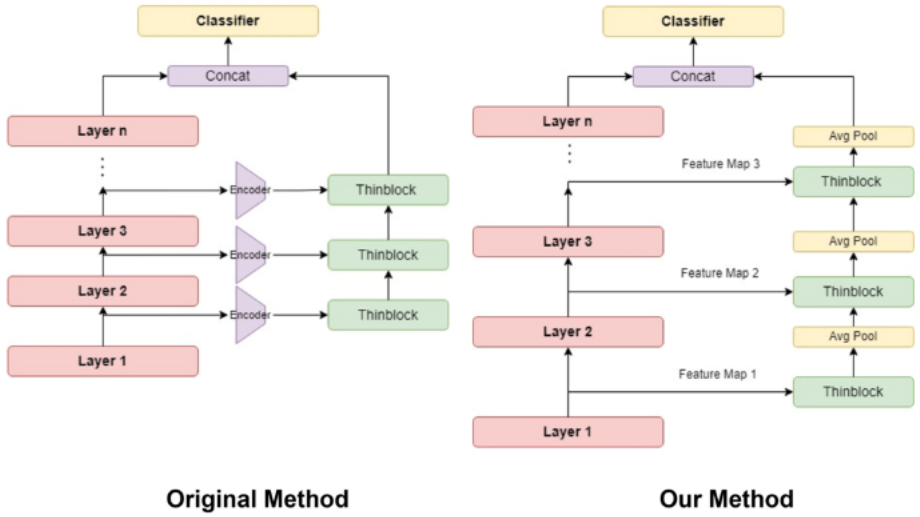

Figure 5.5: UDTA like structure used in this project. The encoder layer is removed to reduce the overall number of parameters in the network while average pooling is used to ensure the output shape of the previous layer matches the shape of the incoming feature map

A similar thinblock architecture is used as seen in figure 5.6 but the initial convolution to increase the number of channels is left out. This is due to the already large number of channels of the feature maps extracted from the BNN. A skip connection is also added before the pointwise convolution. To further minimise the number of weights, the amount of groupings (the number of channels used in an output channel) in the pointwise connection is also varied.

Table 5.3: Comparison of parameters introduced between thinblock proposed by [6] and the thinblock used in this project. Note that $C_d > C_{in}$

	Original Thinblock	Our Thinblock
1x1 Conv	$C_{in}C_d$	-
2D BatchNorm	$2C_d$	-
3x3 DW Conv	$9C_d$	$9C_{in}$
2D BatchNorm	$2C_d$	$2C_{in}$
1x1 Conv	C_dC_{out}	$C_{in}C_{out}$
BatchNorm	$2C_{out}$	$2C_{out}$
Total Parameters	$C_d(C_{in}+C_{out}+13)+2C_{out}$	$C_{in}(C_{out}+11)+2C_{out}$

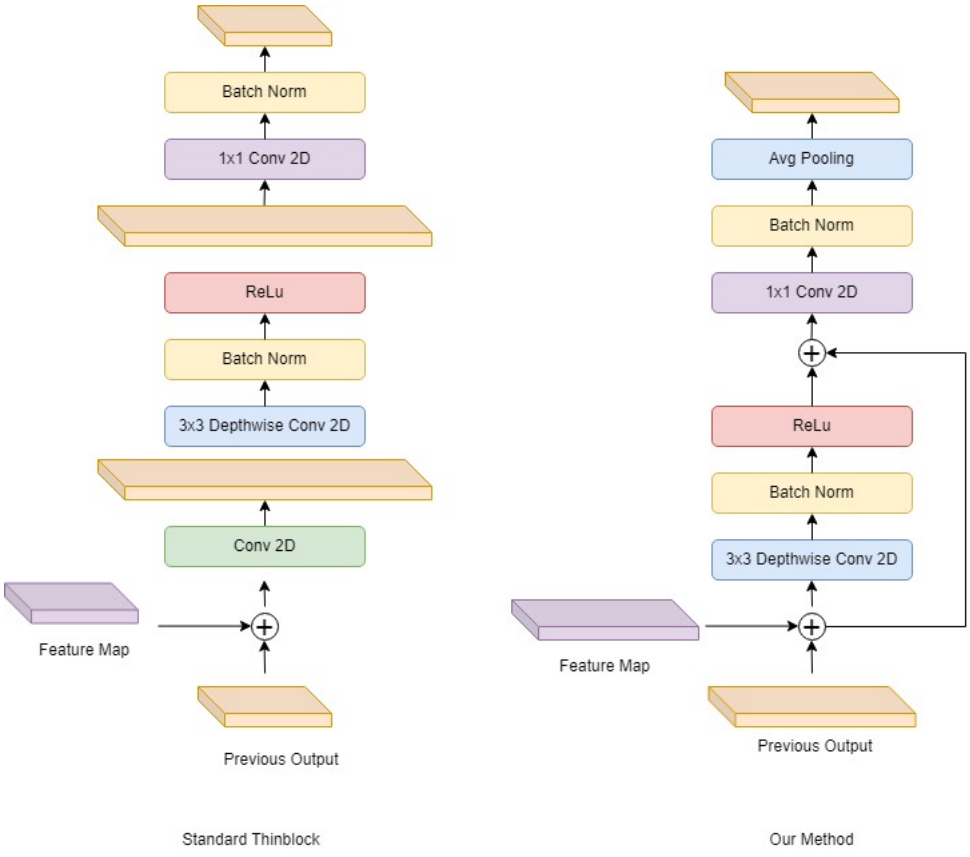

Figure 5.6: Structure of thinblock used in [6] and our thinblock design. Average pooling is added to ensure the output shape matches the feature map of the next layer.

5.5 Reducing Adapter Parameters

Because the base network is composed primarily of 1 bit weights, there is a more stringent constraint on the number of weights one can introduce via adapter networks. This is due to adapter weights being 32-bit compared to their 1 bit counterpart in the base BNN. This project aims to investigate 2 methods to reduce weights: Grouped Convolution and pruning.

5.5.1 Grouped Convolution

A single 2D Convolution filter typically include all input channels C_{in}. This means for a kernel size of (k,k) and output channels C_{out} the total number of parameters introduced is $C_{in}C_{out}k^2$. Instead of using all input channels, one can instead use a subset of the input channels in so called grouped or depthwise seperable convolution. Assuming g is the number of subsets to create (and g divides the C_{in} and C_{out}), then the number of parameters introduced is $\frac{C_{in}C_{out}k^2}{g}$. The extreme case of this is depthwise convolution seen in the thinblock in figure 5.6 where g is equal to the number of input channels (assuming $C_{in} = C_{out}$). In this case each input channel is convolved with its own independent filter.

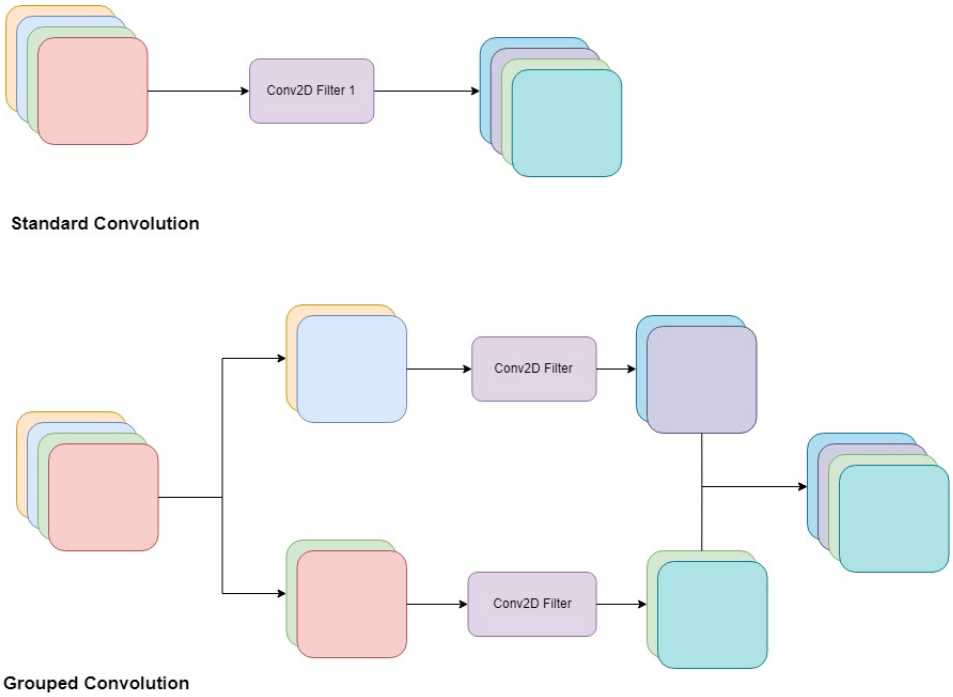

Figure 5.7: Standard convolution vs Grouped or depthwise convolution. For standard convolution, each input channel is included in each convolution filter. For grouped convolution, the input channels are separated into g groups and then regular convolution is applied to each of these groups and the output channels are concatenated together

This project investigates the effect of increasing g on the training accuracy for both serial and parallel adapter methods. For the thinblock adapter, grouped convolution is applied to the final 1x1 convolutional layer (weights of this layer are then $C_{in}C_{out}/g$.

5.5.2 Pruning

Another method for reducing parameters in adapters is by pruning.Pruning in neural networks is an network optimisation technique where weights are removed from the neural network. Researchers have shown that up to 90% of weights can be pruned/removed without affecting test accuracy [48]. Pruning strategies can range from being completely random to a global strategy where the smallest $p\%$ of weights are removed.

By pruning the network of unnecessary weights, the resulting network can be smaller and be made to faster allowing networks to be better deployed on lower end hardware. In this project, one shot pruning is employed with a mixture of structured and unstructured pruning.

Pruning in Pytorch is implemented by applying a bitmask over the original weights setting them to zero [26]. As such, no actual speed up and weights saving is seen after pruning said weights. However, the strategy presented would allow one to transfer said

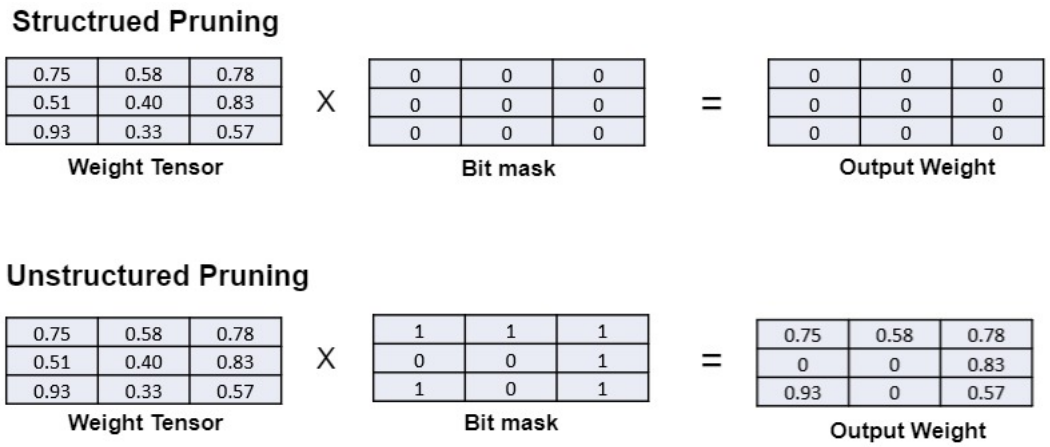

Figure 5.8: Example of how pruning is implemented in Pytorch where a bitmask is used to set 'pruned' weights to zero. For structured pruning the shape of the weight tensor is considered so either the entire weight tensor is pruned or not pruned. For unstructured pruning, any parameter meeting the pruning criteria (e.g. smallest parameter magnitude) is pruned.

weights to a different layer that would allow one to see tangible speed up and weight savings as seen in figure 5.9

5.5.2.1 Pruning Strategy

The pruning strategy is applied thin block structure in figure 5.6 and thus only the parallel adapter network is pruned for this project. A one shot pruning method is performed whereby the adapter network is trained with all parameters during the first half of training and then pruned and then further trained.

Most parameters in a CNN are located at the convolutional layers due to the number of parameters due to the number of parameters being $O(C_{in}C_{out})$. Structured pruning on the filter level is performed on the depthwise convolution layer. The L1 norm is used to determine the magnitude of the matrix and the smallest p% of filters are removed.

$$||W||_{L1} = \sum_i \sum_j |W_{ij}|$$

Where $|W_{ij}|$ is the absolute value for element W_{ij}. For the pointwise convolution layer, an unstructured pruning approach is used where the smallest $p\%$ parameters are removed. Unstructured pruning is used for the pointwise convolution as each filter is equivalent to outputting a linear combination of input channels and so the pruned filter can be transferred to a new layer.

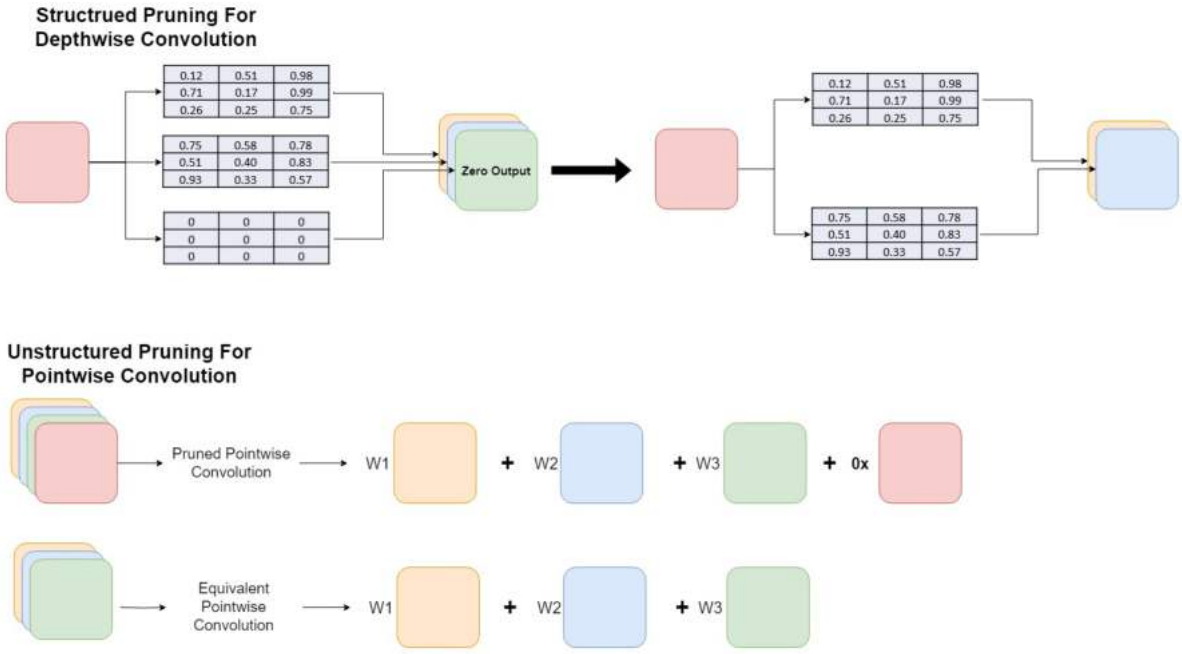

Figure 5.9: Example of how pruned layers can be transferred to a new layer and truly be pruned. Using the pruning strategy above, the new layers would simply exclude the corresponding input channel during convolution

.

5.6 Benchmarks

5.6.1 Cifar5-5 and Cifar80-20

The STE BNN needs to first be trained from scratch. However, this inviestigation requires an additional dataset to act as the domain to finetune the base network. As such, the Cifar10 and Cifar100 datasets are split into two datasets, one serving as the initial training dataset and the second acting as the new target domain to finetune the trained network to.

For Cifar5-5, the first 5 alphabetical classes are used as the initial training dataset for the STE BNN. The remaining 5 classes are then used to create the transfer learning benchmark as seen in figure 5.10. The BNN is trained until comparable accuracy is reached compared to its fp32 couterpart [1]. This trained network is then finetuned on the second dataset.

The Cifar80-20 benchmark also follows a similar process, however the STE BNN is first trained on the last 80 classes in Cifar100. Once trained, the remaining 20 classes serve as the finetuning benchmark.

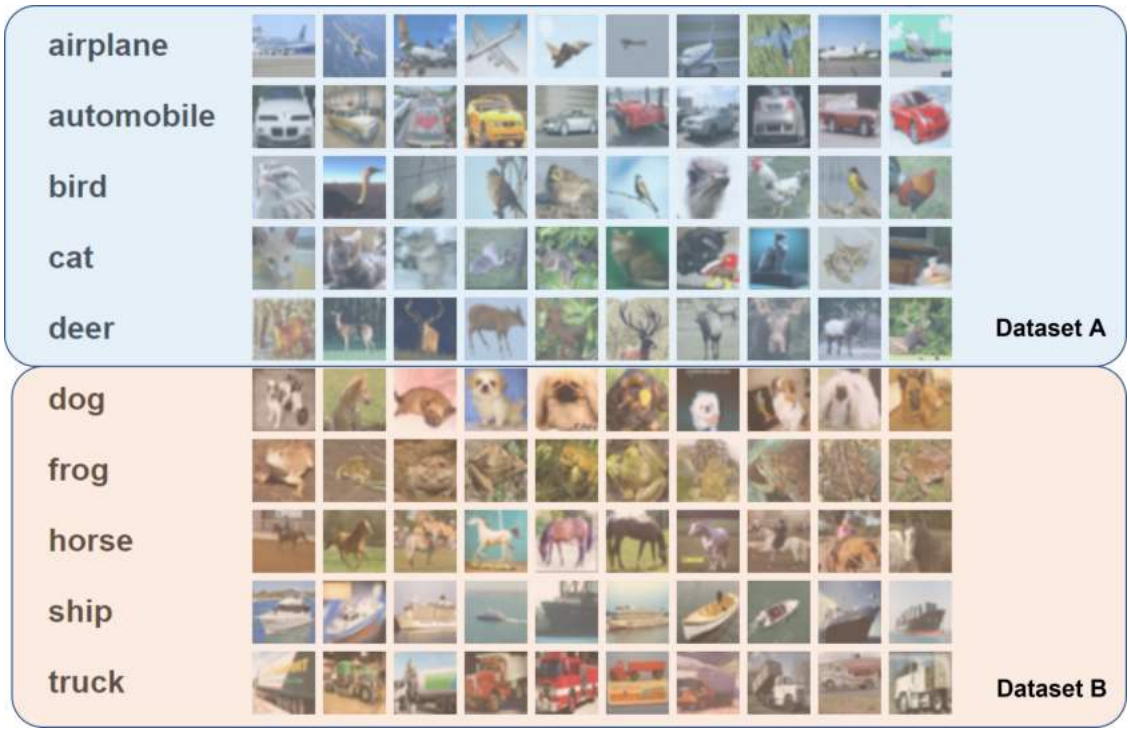

Figure 5.10: Sample of 32x32 images of each class in the Cifar-10 Dataset. Each class contains 5000 training images and 1000 test images [23]. For the initial investigation, the Cifar-10 dataset is split into dataset A and B. The model is first trained on dataset A and then finetuned onto dataset B

5.6.2 Flowers102 and Oxford Pets Dataset

Cifar10 and Cifar100 provide a fast initial dataset for experimentation due to it small 3x32x32 resolution. To better test adapter networks for finetuning, datasets containing images of much higher resolution (e.g. 224x224 RGB images) are used. In particular, the Flower102 dataset (102 flower catagories) and Oxford Pets (37 different breeds of dogs and cats) are used as examples of domains one might wish to finetune a pretrained network for classification [24, 25].

Table 5.4: Dataset Information for flower102 and Oxford Pets dataset. The image size varies between images and number of images for each class is not equal

	Classes	**Training Images**	**Test Images**	**Total Images**
Flowers102	102	1020	1020	2040
Oxford Pets	37	3680	3669	7349

ImageNet is a popular benchmark for image classification containing over 14 million images and 1000 classes [56]. Model that achieve high accuracy on this dataset are often

used as pretrained models to be finetuned to other domains [16, 6, 5]. Due to the size of ImageNet, significant computational resources or time are required to train a network. As such, for the Flowers102 and Oxford Pets dataset, a pretrained Resnet18 ReActNet BNN is used as the base BNN [4].

Figure 5.11: Top Row: Images from Flower Dataset [24]. Bottom Row: Sample images from Oxford Pets dataset [25]. Note that image resolution can vary and is not fixed

5.7 Training Procedures

5.7.1 Cifar 5-5 and Cifar 80-20

5.7.1.1 Data Augmentation

The following data augmentations were used: The means and standard deviations are derived

Table 5.5: Data Augmentation for Training on Cifar5-5 and Cifar80-20 benchmarks. The Mean and std are derived from initial 80 classes used in the initial training of the BNN

Data Augmentation	About
ToTensor()	0-255 range is normalised to [0,1]
RandomCrop()	Random crop of Image with padding = 4
Random Horizontal Flip	Flip the image horizontally with p = 0.5
Normalize	Normalize image channels with mean: (0.5.0.485,0.442) std: (0.268, 0.256, 0.277)

after the images are rescaled to $[0, 1)$ and are calculated from the initial dataset (e.g. the 80 classes in Cifar100)

5.7.1.2 Hyperparameters

For the hyperparameters a learning rate of 1e-3 was used in conjuction with a batch size of 64. The Adam Optimizer is used with default parameters [57]. The BNN is first trained for 200 Epochs on the initial dataset (e.g. Dataset A in figure 5.10). The BNN used as the base network for finrtuning is the BNN that achieves the highest top 1% accuracy during the training.

For adapter training the learning rate,batch size and Adam optimizer remains the same as above unless specified. For standard finetuning methods, the learning rate is reduced to 1e-4.

5.7.1.3 Finetuning Procedures

The model is first finetuned to the target domain using the standard methods where only the head/classification layer is retrained while keeping all other weights frozen and the 'standard' finetuning method where all weights are retrained on the the target domain. These methods serve as benchmarks to measure the effectiveness of using adapters for finetuning.

5.7.1.4 Cifar 5-5 Experiments

For adapters , all weights in the base BNN class are frozen except for the final linear/classification layer(s). For Cifar10, only serial adapters are examined. 4 serial adapter configurations are performed: a single adapter placed after each residual block as seen in figure 5.1 and an experiment where adapters are placed after all 3 layers. The results are then compared with standard finetuning techniques.

5.7.1.5 Cifar 80-20 Experiments

Both the serial approach in the Cifar 10 experiments and UDTA approach are analysed with the Cifar 100. Furthermore, for the experiment whereby an adapter is added after every residual block, the effect on test accuracy due to grouped convolutions on the is also investigated. Grouped convolutions of 1,2,5,10 and 20 groups are investigated.

For the UDTA approach, feature maps are extracted after every residual block. Both pruning and grouped convolutions are investigated as methods to reduce parameter weights. Grouped convolutions is only applied to the final pointwise convolution layer seen in figure 5.6. Groups of 1,2,5,10,20 and groups equal to the number of input channels to each layer are investigated. The oneshot pruning strategy discussed in section 5.5.2 and is applied to each adapter in the UDTA network. The effect of pruning is examined at a pruning rate of 0,80,90,95 and 99%.

5.7.2 OxfordPets and Flowers102 Dataset

5.7.2.1 Data Augmentation

The following data augmentations were used: The means and standard deviations are derived

Table 5.6: Data Augmentation for Training On Pets and Flower Dataset. The Mean and STD are derived from ImageNet

Data Augmentation	About
ToTensor()	0-255 range is normalised to [0,1]
Random Rotation	Randomly rotate image up to 15 degrees
Random Resized Crop	Random crop of Image of size (224,224)
Random Horizontal Flip	Flip the image horizontally with p = 0.5
Random Vertical Flip	Flip the image vertically with p = 0.5
Normalize	Normalize image channels with mean: (0.485.0.456,0.406) std: (0.229, 0.224, 0.225)

after the images are rescaled to $[0, 1)$ and are calculated from the initial dataset (e.g. the 80 classes in Cifar100)

5.7.2.2 Hyperparameters

As a pretrained model is used, no initial training is needed. For adapter training, the hyperparameters follow for both flower102 and Oxford Pets datasets use the same hyperparameters as used in Cifar100 investigations. Finetuning training occurs over 75 epochs.

5.7.2.3 Finetuning Procedures

Adapter training for the flower102 and Oxford Pets dataset follow only focuses on pruning as the main form of weight reduction strategy. This is because of the ability to better control the amount of weights that are removed from the adapter networks. Pruning rates of 80,90,95 and 99% are examined.

Finally, these datasets are used to examine if one needs to concatenate the output layer of the UDTA adapter with the final layer of the base BNN. Concatenating the final layers increases the number of parameters introduced in the final classification layer. This is especially true for the ReAct Net which uses a full precision linear layer as the classification layer [4]. To investigate the effect the same model is also trained with the same procedures but with the final classification layer attached to final layer of UDTA only. If the output shape of the UDTA and base BNN are the same, then only using the UDTA head ensures that no additional parameters are introduced in the final classification layer.

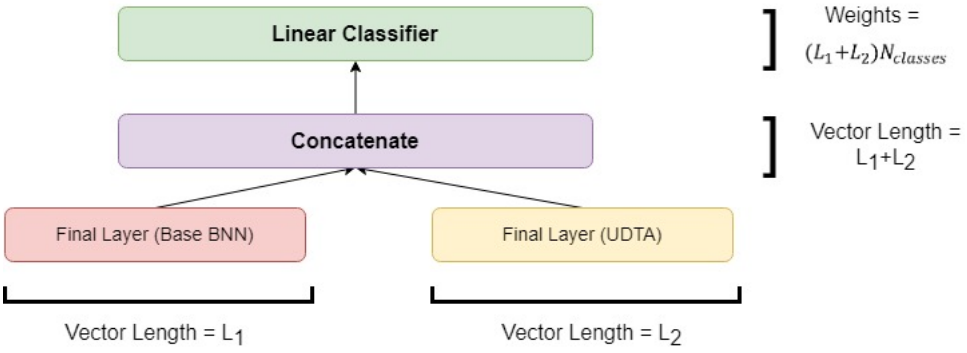

Figure 5.12: Parameters in the final layer are proportional to the concatenated length of the final layers of the base network and and the adapter network. As such using only the final layer of the UDTA end could significantly reduce the number of parameters introduced

5.8 Summary of Methodology

Benchmark	Base BNN	Adapter Strategy	Weight Reducing Strategy
Cifar 5-5	STE BNN	Serial	Using a single adapter + Grouped Conv
Cifar 80-20	STE BNN	Serial + Parallel	Pruning and Grouped Conv
Flower102	ReAct BNN	Parallel	Pruning and UDTA Head only
Oxford Pets	ReAct BNN	Parallel	Pruning and UDTA Head only

Table 5.7: Summary of training procedures investigated in this project

Chapter 6

Results and Discussion

6.1 Cifar5-5 Benchmark

6.1.1 Initial Training

The BNN used for training is trained for 400 epochs on the first 5 classes of Cifar10. It is trained until it reaches comparable accuracy to the baseline Resenet model as seen in table 6.1

	Top 1% Accuracy	**Binary Weights**	**FP32 Weights**	**Total Number of Weights**	**Effective Memory Size (kB)**
BNN	98.72%	4330165	6250	4336415	566
FP32 Resnet18	97.1%	-	175258	175258	701

Table 6.1: Top 1% Over first 5 classes of Cifar10 and comparative memory size for the STE BNN by [1] and the baseline real valued resnet18 like model

6.1.2 Serial Adapter

The last 5 classes in Cifar10 are then used as target dataset to finetune our BNN towards. A serial adapter approach is implemented for these 5 classes. The adapter used is based on [5] and can be seen in figure 5.4. The first set of experiments adds adapters after each residual block as seen in figure 5.1 resulting in 3 adapters being inserted into the network. For this setup, a grouped convolutions at groups of 10 and 20 are also examined. The second set of experiments add only a single adapter after each layer.

6.1.3 Discussion

Regular finetuning method outperforms serial adapters on the remaining Cifar5 classes reaching an accuracy of 96%. The best method of 3 adapters with no grouped convolutions achieves only 91%. While this may appear close and using only a fraction of total weights,

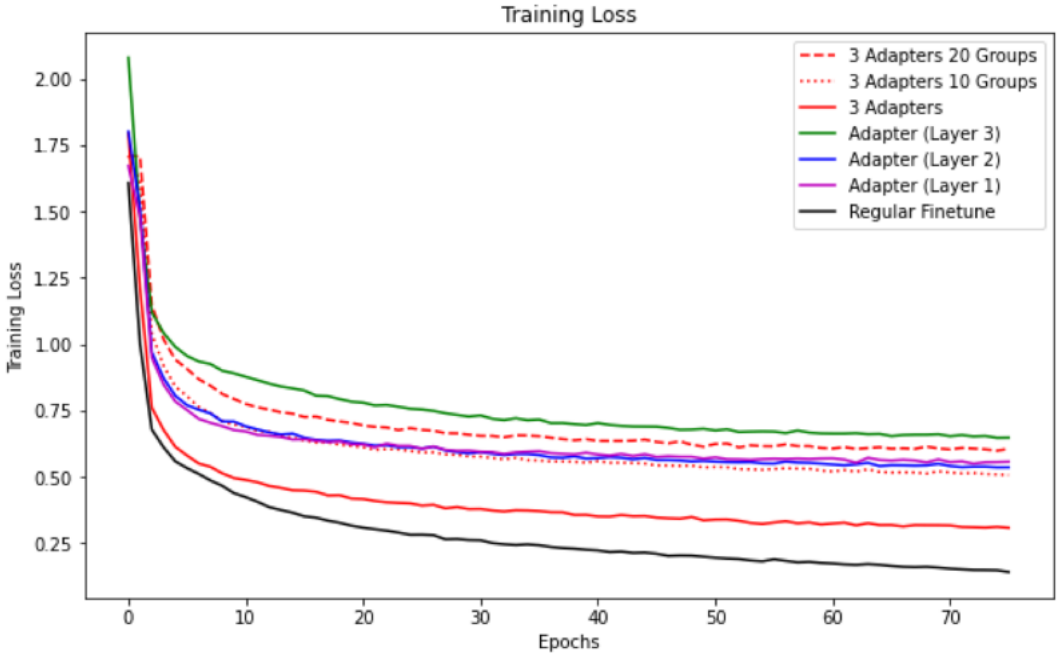

Figure 6.1: Training Loss for various adapter strategies for BNN over the last 5 classes in Cifar10. The Regular fine tuning where all original weights are trained outperforms the use of adapters.

Table 6.2: Top 1% Accuracy For Finetuning on latter half of Cifar5. While Regular Finetuning produces high accuracy results, the average epoch time (training + validation) was faster for the adapter methods. For the fraction of base BNN, the memory size of binary bits is also taken into account

	Top 1% Accuracy	**Weights Introduced**	**Fraction of Base BNN Memory***	**Average Epoch Time (s)**	**Relative Speed Up**
Regular Finetune	96.06%	-	-	33.6	-
Head Only	74.44%	-	-	18.4	1.45
3 Adapters	91.04%	135025	95.3%	24.65	1.27
3 Adapters 10 Groups	83.56%	14065	10%	20.9	1.38
3 Adapters 20 Groups	80.04%	7345	5.2%	19.3	1.43
Adapter (Layer 3)	78.34%	102065	72%	19.2	1.43
Adapter (Layer 2)	83.40%	24625	17%	20.0	1.40
Adapter (Layer 1)	82.19%	5105	3.6%	21.0	1.38

if one considers that most of the weights in the BNN are binary, the 3 adapter approach effectively is doubling the size of BNN in terms of memory. This defeats the purpose of adapters being lightweight and 'small' in comparison to the base BNN as one might as well run the baseline real valued Resnet18.

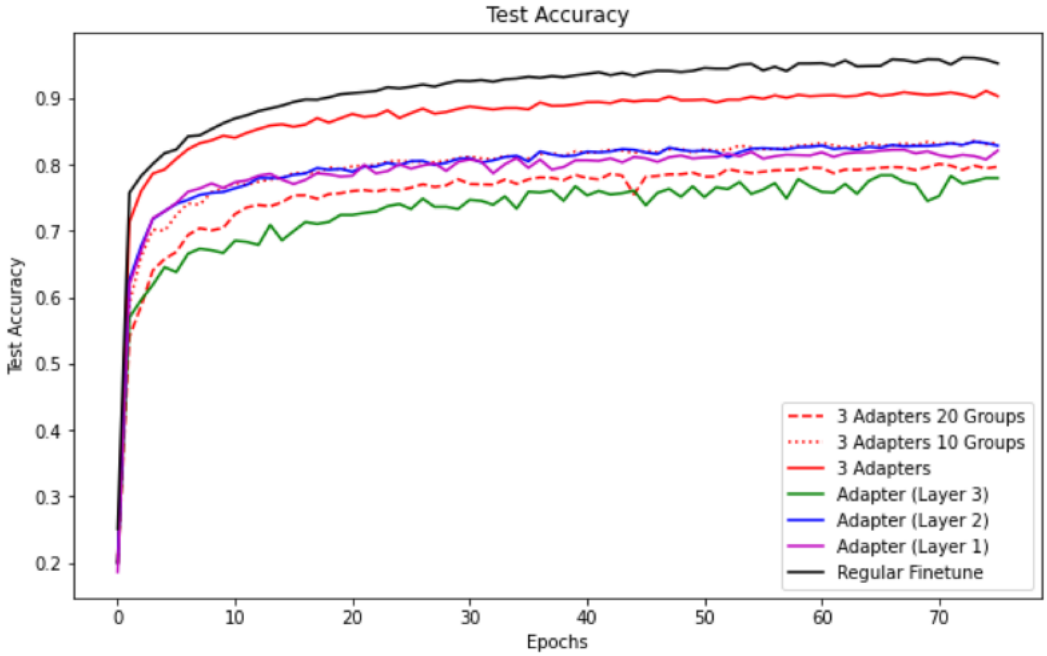

Figure 6.2: Test Accuracy for various adapter strategies for BNN over the last 5 classes in Cifar10. The Regular fine tuning where all original weights are trained outperforms the use of adapters.

When convolutions are replaced by grouped convolutions however, the weights introduced become more reasonable. Using a groups of 10 and 20 for convolution at each adapter, the fraction of BNN memory is 10% and 5% respectively and achieving a smaller accuracy of 83.5 and 80% respectively. Compared to single adapters, using 3 adapters after each layer but with grouped convolutions is more effective. Having 3 adapters with 10 convolution groups achieves comparable accuracy with having a single adapter after layer 2, while having almost half as many weights.

In terms of epoch training time (time for training + validation), regular finetuning is slower than the adapter approach. This is because for regular finetuning, backpropagation must update all weights. This can cost can be increased by BNNs which often have unique approaches to training weights (for example the base BNN for Cifar10 uses the STE approach[1]) [52, 4, 20]. Adapter training sees a 27 to 40% increase in training speed compared to regular finetuning as only the weights in the adapters need to be updated.

In terms of the results above, adding an adapter after each residual block and then using grouped convolutions to reduce the number of introduced parameters would be the best tradeoff considering training accuracy, training time and parameters introduced. Furthermore, this approach of adding an adapter after each layer eliminates the need to find where to place the best adapters. As seen in the table 6.2, placing an adapter after layer 1 would be the best in terms of an accuracy to parameter trade off. However, finding the best location is not

trivial and would require testing the adapter after each layer or block. While this network has 3 blocks/layers, this issue could become problematic for deeper networks.

6.2 Cifar 80-20

6.2.1 Serial Adapter

Table 6.3: Top 1% Accuracy and Number of weights introduced for the serial adapter approach

Finetuning Method	**Top 1% Accuracy**	**# Weights Introduced**	**% of Base BNN Memory**	**Average Epoch Time (s)**	**Relative Speed Up**
Regular Finetuning	75.0%	-	-	13.1	-
Head Only Finetune	65.5%	-	-	7.7	1.41
3 Adapters	76.2%	136640	96.4%	9.2	1.30
3 Adapters (2 Groups)	75.2%	69440	49%	9.0	1.31
3 Adapters (5 Groups)	74.0%	29120	20.5%	9.0	1.31
3 Adapters (10 Groups)	72.6%	15680	11.1%	9.1	1.31
3 Adapters (20 Groups)	70.0%	8960	6.3%	9.2	1.30

6.2.2 UDTA Adapter

6.2.2.1 Pruning Results

Table 6.4: Finetuning Results on the 20 classes in Cifar80-20 and the effect of using Pruning. Number of weights includes the additional weights from the last classification layers

	Top 1% Accuracy	Number of fp32 weights	Number of Binary Weights	% of Base BNN memory	Average Epoch Time (s)	Relative Speed Up
Head Only	65.4%	0	0	0.0%	7.65	1.42
Regular Finetune	75.0%	0	0	0.0%	13.24	1.0
Prune 99%	67.9%	4438	6410	3.3%	8.75	1.34
Prune 95%	71.2%	11292	6410	8.1%	8.73	1.34
Prune 90%	73.4%	19864	6410	14.2%	8.74	1.34
Prune 80%	74.1%	37008	6410	26.2%	9.02	1.32
No Pruning	76.4%	174160	6410	123.0%	9.08	1.31

6.2.2.2 Grouped Convolution

6.2.3 Discussion

Using regular finetuning, the BNN reached an accuracy of 75% over the target domain of 20 different Cifar80-20 classes. Regardless of adapter approach, the accuracy increases

Table 6.5: Finetuning Results on the 20 classes in Cifar80-20 and the effect of Using Grouped Convolution. Number of weights includes the additional weights from the last classification layers

	Top 1% Accuracy	Number of fp32 weights	Number of Binary Weights	% of Base BNN memory	Average Epoch Time (s)	Relative Speed Up
Head Only	65.4%	0	0	0.0%	7.65	1.42
Regular Finetune	75.0%	0	0	0.0%	13.24	1
Grouped Conv All%	70.2%	8560	6410	6.1%	8.44	1.36
Grouped Conv Group 20	72.2%	16080	6410	11.4%	8.56	1.35
Grouped Conv Group 10	73.2%	24400	6410	17.3%	8.57	1.35
Grouped Conv Group 5	74.5%	41040	6410	29.0%	8.54	1.35
Grouped Conv Group 2	75.8%	90960	6410	64.2%	8.52	1.25
No Grouped Convolution	76.4%	174160	6410	123.0%	9.08	1.31

depending on the number of domain specific parameters added to the network. For example using grouped convolution where there are C_{in} groups in the UDTA, and serial adapters with 20 groups both introduce approximately the same number of parameters and achieve the same accuracy of about 70 %. By adding parameters equal to or exceeding the base BNN's memory, adapters can achieve higher accuracy than regular finetuning however, this would defeat the purpose of adapter approach. This can be seen in figure 6.3 where all adapter strategies achieve similar performance depending on the number of parameters introduced. The accuracy is approximately proportional to the logarithim of the number of adapter parameters introduced.

All three adapter strategies see a faster training time compared to regular finetuning with each epoch being approximately 1.3-1.4 times faster than regular finetuning which must update all weights in the base BNN. As the binary weights are treated as real valued weights during back propagation, the number of gradient updates are 2-3 orders of magnitude more than when using adapter strategies.

Based on the training speed and accuracy vs parameters seen in figure 6.3, any adapter strategy presented may be a reasonable approach. For ease of use, the serial approach would be recommended as it does not need modification of the underlying network class. In Pytorch, one can append the adapter to the end of an existing layer. UDTA approach requires the extraction of intermediate features and so would require a rewriting of the class. This can be cumbersome especially for deeper networks.

For flexibility, pruning would be the best strategy over using grouped convolutions as it is more flexible in terms of how many parameters to have in the adapter as the number of groups g in grouped convolution must be divide C_{in} or C_{out} and $g_{max} = \min(C_{in}, C_{out})$. Furthermore, as pruning is assessed during training time, pruning could be better tailored to specific datasets. Grouped convolution groups adjacent channels together which may not be optimal.

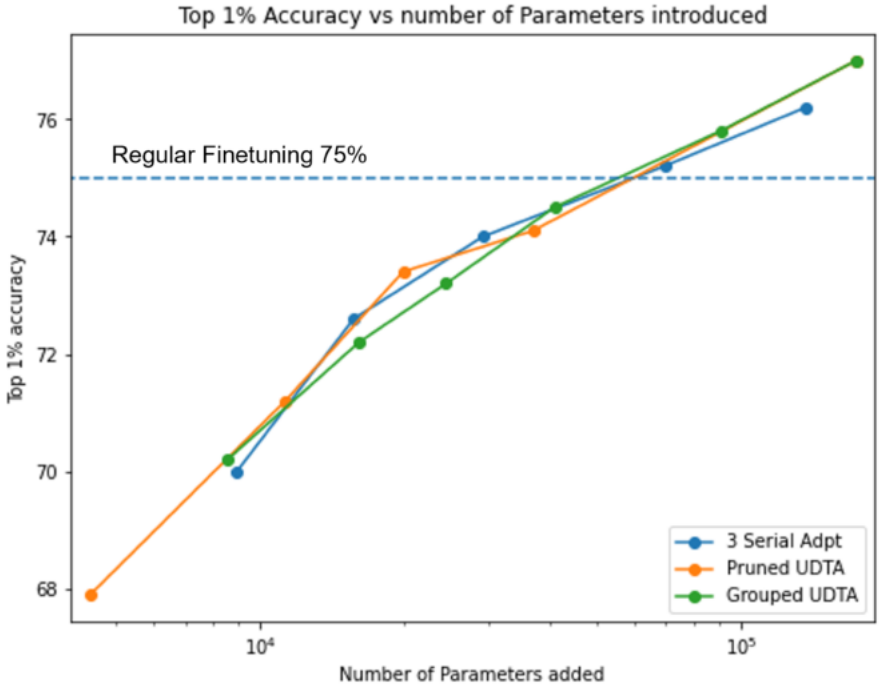

Figure 6.3: Comparison of Accuracy based on the number of weights introduced. The accuracy change is similar between the 3 adapter strategies based on the number of domain specific weights

For training convergence, the grouped convolution combined with UDTA adapters may be a good compromise between speed of training and accuracy. As seen in figure 6.4, even using 20 groups, the adapter converges to it's max accuracy in the first 10 epochs. Regular finetuning reaches the 75% accuracy at the end of the 75 epochs. This also means one can prune much earlier perhaps after only 10 epochs using the UDTA approach.

A compromise between number of parameters introduced an accuracy loss would be approximately 11,000 - 16,000 weights or 8% - 11% of the base BNN's memory which would see only a 2.5% - 4% decrease in accuracy.

6.3 Flowers 102 Dataset

6.3.1 Discussion

For the flower102 dataset, regular finetuning achieved an accuracy of 87% while adapters achieved an accuracy of 80%-82.8% with UDTA adapters even with 99% pruning.

Because of concatenation of the final UDTA layer and the final layer of the base BNN, an additional $(512) \times 102 = 52,224$ weights are introduced. This alone accounts for an additional 9% of weights relative to the base BNN memory. As seen in table 6.6, even at a pruning rate of 99%, the total number of introduced parameters is still 10% relative to

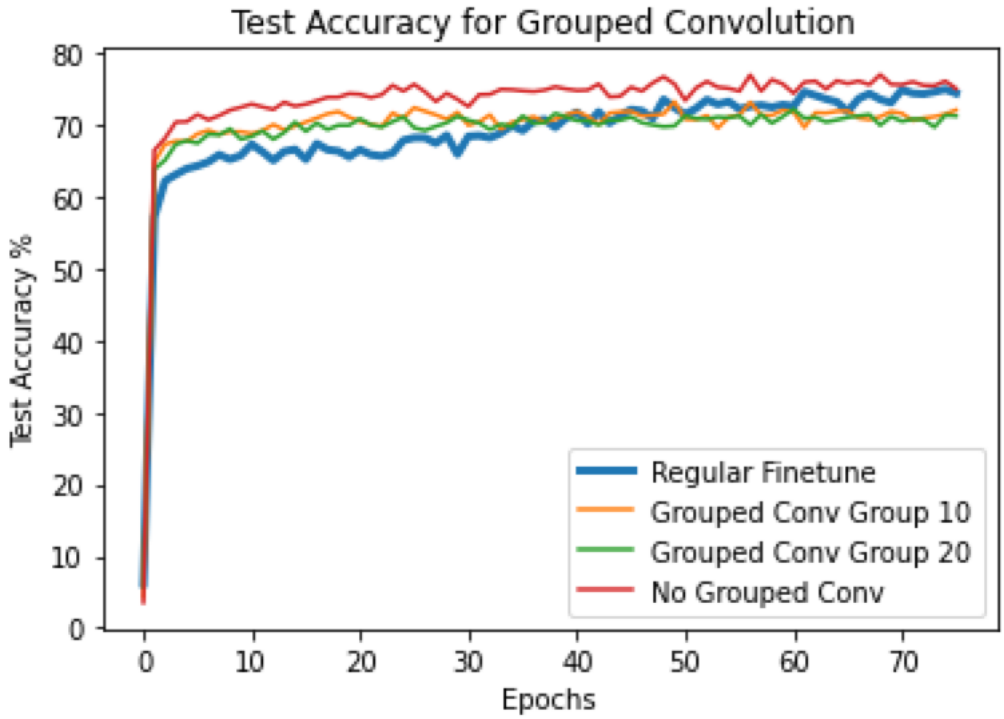

Figure 6.4: Test Accuracy over epochs during finetuning. Note that even with 20 groups, grouped convolution converges faster than regular finetuning. Note that No Grouped Convolution is equivalent to No pruning.

Table 6.6: Flowers102 Finetuning Results and the effect of using Pruning and Only using adapter network for output

	Top 1% Accuracy	Number of fp32 weights	% of Base BNN memory	Average Epoch Time (s)	Average Epoch Time (s)
Regular Finetune	87.0%	0	0.0%	49.05	1
Prune 99%	83.6%	61443	10.2%	48.35	1.01
Prune 95%	85.2%	79152	13.2%	48.35	1.01
Prune 90%	85.1%	101292	16.8%	48.38	1.01
Prune 80%	85.3%	145574	24.2%	48.36	1.01
No Pruning	85.3%	499827	83.1%	48.4	1.01
Prune 99% (Adpt Head)	81.5%	9168	1.5%	48.53	1.01
Prune 95% (Adpt Head)	83.8%	26877	4.5%	48.44	1.01
Prune 90% (Adpt Head)	83.8%	49017	8.1%	48.41	1.01
Prune 80% (Adpt Head)	84.3%	93299	15.5%	48.38	1.01
No Pruning (Adpt Head)	84.9%	447552	74.4%	48.31	1.02

memory. By using only the final layer of the adapter for the input to the linear classification layer (called Adpt Head in the table), one can further reduce the number of parameters. Comparing between the same amount of pruning, using only the adapter for the final output layer decreases the accuracy by 1-2% while decreasing the amount of added weights by 8.7%

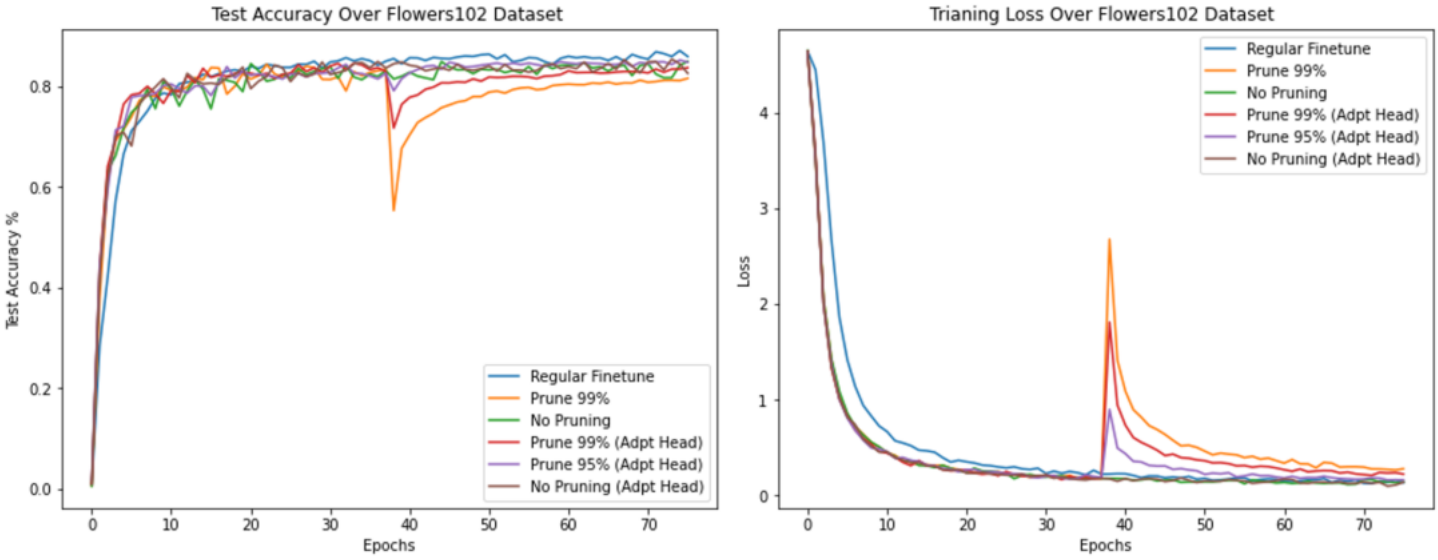

Figure 6.5: Sample plot of various finetuning runs over Flower102 dataset. Note the drop in accuracy is due to pruning occurring half way through the total number of epoch (75/2 = 37). Note that the training loss for regular finetuning is consistently higher than the adapter method for the first half of training

of the Base BNN's memory. Ultimately, a decrease of 5% in accuracy may be worthwhile if only weights worth 1.5% of the Base BNN memory are needed.

Interestingly, one see that that the training loss for regular finetuning in figure 6.5 is higher than when using the adapter approach. This was also seen with Cifar100 where, the regular finetuning required more training time to reach saturation compared to adapters even when a low number of parameters were added as seen in figure **??**. This indicates adapters may be able to train in fewer epochs compared to regular finetuning.

6.4 Oxford Pets 102 Dataset

From table 6.7 finetuning the head only actually yielded the best resut at 76.8% followed by regular finetuning at 75.5%. This suggests that the main factor governing the accuracy over the new domain in the BNN is the final linear classification layer. This is intersting as if one examines the training loss for head only in figure 6.6, it is consistently high with 99% pruning reaching similar training loss by 75 epochs.

The results over the Oxford Pets 102 Dataset yield similar results to the flowers 102 results. However, as there are only 37 classes to classify, using only adapter output introduces only $(512) \times 37 = 18,944$ weights or 3.2% of the Base BNN memory. As such, using only the adapter head is more useful choice when needing to classifying a large number of classes.

Table 6.7: OxfordPets Finetuning Results and the effect of using Pruning and Only using adapter network for output

	Top 1% Accuracy	Number of fp32 weights	% of Base BNN memory	Average Epoch Time (s)	Average Epoch Time (s)
Head Only Finetune	76.80%	0	0.00%	44.46	1.1
Regular Finetune	75.50%	0	0.00%	49.15	1
Prune 99%	74.80%	28130	4.70%	45.41	1.08
Prune 95%	74.90%	45839	7.60%	44.36	1.1
Prune 90%	75.10%	67979	11.30%	45.29	1.08
Prune 80%	74.50%	112261	18.70%	45.37	1.08
No Pruning	74.70%	466514	77.50%	45.53	1.07
Prune 99% (Adpt Head)	71.40%	9168	1.50%	43.87	1.11
Prune 95% (Adpt Head)	74.80%	26877	4.50%	42.61	1.13
Prune 90% (Adpt Head)	75.50%	49017	8.10%	42.62	1.13
Prune 80% (Adpt Head)	74.40%	93299	15.50%	42.65	1.13
No Pruning (Adpt Head)	74.20%	447552	74.40%	42.8	1.13

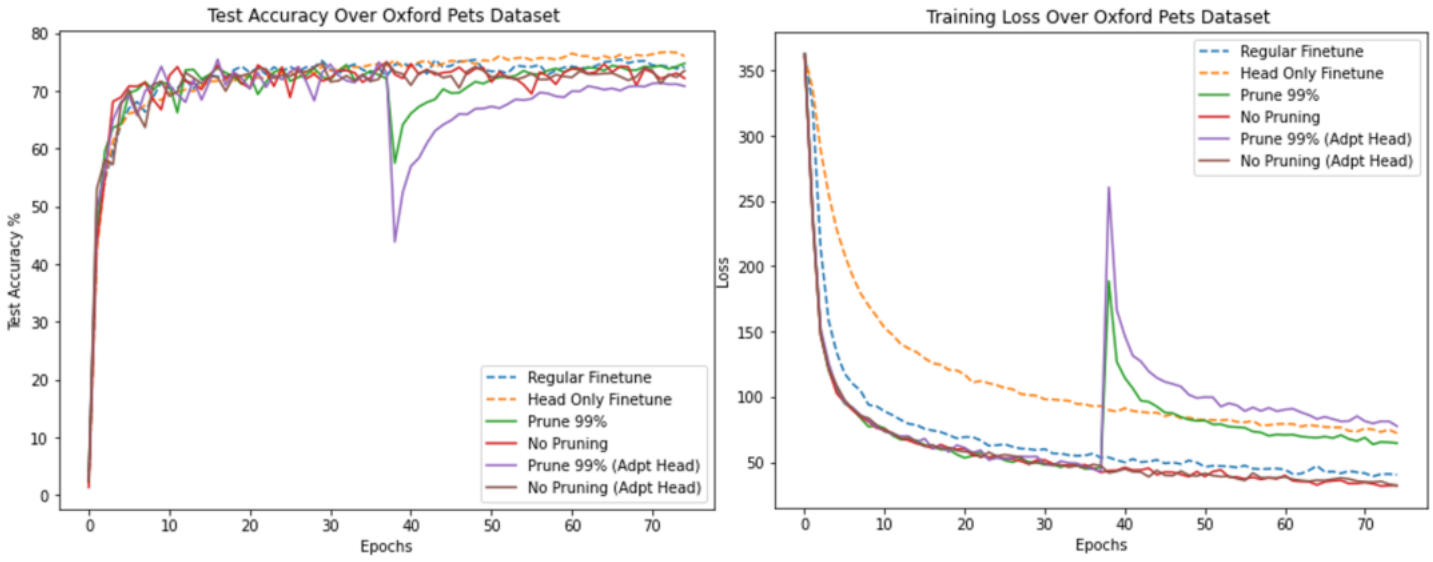

Figure 6.6: Sample plot of various finetuning runs over OxfordPets dataset [25]. Note that although finetuning the head only achieved the highest accuracy, the training loss is the second highest after 75 epochs of training

Chapter 7

Future Work and Improvement

7.1 Finetuning Methods

7.1.1 Adapter Architecture

This capstone project investigated the use adapters used by [5] for serial adapters and the thinblock design for the UDTA adapter [6]. Further investigations into different architectures could help make adapters more effecient and reliable over different domains. In particular, this capstone project used average pooling after an adapter block to ensure the output size would match the incoming feature. A more novel way of approaching this issue may further help adapter accuracy.

7.1.2 LoRA

Adapters were first proposed for the use in NLP [21] as a parameter efficient way of finetuning large language models. However, adapters can be difficult to use effectively as they introduce additional design parameters such as the adapter architecture and where to actually place said adapter [58]. Furthermore, adapters introduce additionally parameters and inference time [58].

The LoRA (Low Rank Adaptation) method has seen large success in recent time in quickly finetuning large language models, achieving SOTA finetuned accuracy on pretrained language model with a single GPU [58]. Using LoRA, Taori et al finetuned a 7 billion llama LLM to create the Alpaca LLM in less than an hour of GPU training across 8 Nvidia A100s [10, 11]. Finetuning consisted of only 1.2 million training parameters or 0.017% of total parameters. However, these additional parameters do need introduce any additional inference time as they adjust the existing weights [58] when loaded in.

The LoRA method freezes the pretrained weights and introduces a low rank matrix decomposition as the target weights to train under the hypothesis that weights updates during finetuning have a low 'intrinsic rank' [58].

$$W_{finetune} = W_0 + \Delta W$$

Where W_0 are the frozen pretrained weights of size $\mathbb{R}^{d\times k}$ and $\Delta W = BA$ are the trainable weights. Here B is a matrix of size $\mathbb{R}^{d\times r}$ and A is a matrix of size $\mathbb{R}^{r\times k}$, with r being the *rank* and $|r| << \min(d,k)$.

The advantages of LoRA is that it has the same ability as parameters to be easily swapped out depending on the target task but achieving this by not introducing any new additional parameters as the ΔW is directly added to the pretrained weights.

LoRA currently is targeted for finetuning LLM which typically use transformer based models. In the future applying LoRA-like methods to CNNs and quantized models could be interesting as this could eliminate the need to introduce additional parameters reducing the computational resources to run a model on low powered devices.

7.2 Reducing Computing Resources

7.2.1 Quantization

For this project, adapter weights were full 32 bit floating point numbers. To further reduce computational resources, quantization could be used to further reduce the additional memory overhead and also allow faster inference times. Quantization converts the floating point operations into integer operations increasing inference time and reducing hardware requirements. In the future, it would be interesting to examine the accuracy trade off when quantizing the adapters to 16bit, 8bit and even lower [59].

7.2.2 Pruning

Only oneshot pruning was investigated in this project where the p% of weights are pruned all at once. However [48] showed that iterative pruning, where only $p^{1/n}\%$ of weights were pruned each round and the remaining weights are reset to their original initialisation over n rounds produced more stable and lower loss in accuracy compared to oneshot pruning. Using iterative pruning may yield better pruned adapter networks which can further reduce computational demand and resources.

Furthermore, pruning was not applied to the final classification layer. Pruning the final layer could further reduce the number of parameters especially for the UDTA approach where the final layer of the base BNN and adapter are concatenated together. Currently sparse matrix multiplication is not implement in many deep learning frameworks. To see tangible memory and speed improvements, one would have to use a structured approach, where entire columns in the linear layer are pruned. This is equivalent to removing input features e.g. instead of 512 input features, 75% of column pruning would only use 128 of the total input features.

7.3 On Device Training

Ultimately, the goal of BNNs and adapters is to potentially achieve on-device finetuning. However, the methods presented in this capstone project would not be feasible for on-device training as the adapters must be first over-parameterised and then pruned to achieve both accuracy and weight reduction [48].

However, the iterative pruning could give a good pathway to achieving semi on-device training. Given an initial domain agnostic dataset, one could use the iterative pruning strategies outlined in [48] to identify 'winning lottery tickets' weight initialisations for general image classification. The pruned adapters could then be exported to low powered devices and then either further trained on the device itself.

Chapter 8

Conclusion

This capstone project examined the effectiveness of adapters as a finetuning method for BNN based CNNs. The approaches were derived from existing adapter techniques. However, BNNs provide a much harder constraint on the number of parameters one can introduce as one needs to consider the memory overhead between traditional 32-bit floating point weights and 1 bit binary weights. This meant modifications to existing adapter architectures were needed (such as pruning and grouped convolution), to reduce the size of the introduced adapters [5, 6]. It was found adapters, regardless of strategy, can achieve comparable accuracy, reduce training time while being memory efficient.

Across all datasets examined, the accuracy of adapter based transfer learning was lower than regular finetuning where all weights are trained. However, this difference was only 1-5 percentages points in difference with the highly optimized adapters. Furthermore, this result scaled to more realistic datasets such as Oxford Pets and Flower102, both of which use high resolution images and training samples of less than 100 per class [25, 24]. The loss of accuracy when using adapter finetuning could still be acceptable for devices with low computational resources. This is because adapters provide a memory efficient way of repurposing a pretrained model to a new target domain so multiple adapters, and so target domains, can be stored and swapped out efficiently [22, 5, 21].

The overall training time per epoch was also demonstrably faster when using adapters where training times could be up 1.4x faster per epoch. Results from figures 6.4, 6.5 and 6.6 showed that adapters also reach their maximum accuracy earlier than regular finetuning which often required the maximum number of provided iterations.

Finally, adapters can be made very small being less than 5% of the base BNN's total memory while only losing a few percentage point in accuracy. Parameter reducing techniques such as grouped convolution and pruning were required to reduce the number of parameters in the adapters. Furthermore, it is important to note that as BNN's have a lower theoretical

memory (assuming one has hardware to store single bit weights and activations) than their floating point counterpart, these adapters are much smaller compared to those found in literature, which can be up to 25% of the base network [6] as the 1 bit size of binary weights must be taken in to account when using BNNs. Across the datasets a sweetspot of adapters between 6 and 12% of the base BNN's memory was found compromising between adapter size and accuracy.

8.1 Learned Practices

Empirically, figure 6.3, showed that the adapter accuracy did not depend on the adapter approach used a mainly depended on the number of weights introduced. More complicated datasets would be needed to truly measure this effect

8.2 Limitations of study

The strategies and approaches in this capstone project are rather theoretical in that the binary weights are not truly binary. This means the benefits such as memory saving and reduced hardware complexity are not measured (in reality BNNs such as that from [1] still use floating point arithmetic but are converted to binary weights during inference and training).

Secondly, the pruned weights are not truly pruned but rather set to zero during pruning [26]. Having truly pruned layers would allow one to analyse the speed up as one prunes the network. As such the weight reductions from pruning in this project are purely theoretical.

Finally, this capstone only used a resnet18 BNN variant [4]. Using binarized networks designed to run on lower end hardware such as MobileNet would be more applicability and efficient than using resnet based models [55].

8.3 Future Directions

This capstone project was focused entirely on CNN based BNNs and on image classification. Extending this study to more complicated computer vision tasks such as image segmentation or object detection would increase the applicability of BNNs and adapters.

Furthermore, the development of specialised hardware to run truly binary BNNs would enable better measurement of the improved efficiency of using BNNs instead of full precision networks. Currently the best approximation to speed up when using BNNs would be to convert the binary weights activations into 8-bit signed integers.

Finally, on-device learning would be an interesting direction to examine. The UDTA approach could help facilitate this as backpropagation does not go through the base BNN. Furthermore, UDTA gives high flexibility in the adapter design allowing for highly efficient adapter architectures to be used.

Bibliography

[1] Matthieu Courbariaux et al. *Binarized Neural Networks: Training Deep Neural Networks with Weights and Activations Constrained to +1 or -1*. 2016. DOI: 10.48550/ARXIV.1602.02830. URL: https://arxiv.org/abs/1602.02830.

[2] Mohammad Rastegari et al. *XNOR-Net: ImageNet Classification Using Binary Convolutional Neural Networks*. 2016. DOI: 10.48550/ARXIV.1603.05279. URL: https://arxiv.org/abs/1603.05279.

[3] Brais Martinez et al. *Training Binary Neural Networks with Real-to-Binary Convolutions*. 2020. arXiv: 2003.11535 [cs.CV].

[4] Zechun Liu et al. *ReActNet: Towards Precise Binary Neural Network with Generalized Activation Functions*. 2020. arXiv: 2003.03488 [cs.CV].

[5] Sylvestre-Alvise Rebuffi, Hakan Bilen, and Andrea Vedaldi. *Learning multiple visual domains with residual adapters*. 2017. DOI: 10.48550/ARXIV.1705.08045. URL: https://arxiv.org/abs/1705.08045.

[6] Han Gyel Sun et al. *Unidirectional Thin Adapter for Efficient Adaptation of Deep Neural Networks*. 2022. arXiv: 2203.10463 [cs.CV].

[7] G. Cybenko. "Approximation by superpositions of a sigmoidal function". In: *Mathematics of Control, Signals and Systems* 2.4 (Dec. 1989), pp. 303–314. ISSN: 1435-568X. DOI: 10.1007/BF02551274. URL: https://doi.org/10.1007/BF02551274.

[8] Tom B. Brown et al. *Language Models are Few-Shot Learners*. 2020. DOI: 10.48550/ARXIV.2005.14165. URL: https://arxiv.org/abs/2005.14165.

[9] David Silver et al. *Mastering Chess and Shogi by Self-Play with a General Reinforcement Learning Algorithm*. 2017. DOI: 10.48550/ARXIV.1712.01815. URL: https://arxiv.org/abs/1712.01815.

[10] Rohan Taori et al. *Stanford Alpaca: An Instruction-following LLaMA model*. https://github.com/tatsu-lab/stanford_alpaca. 2023.

[11] Hugo Touvron et al. *LLaMA: Open and Efficient Foundation Language Models*. 2023. arXiv: 2302.13971 [cs.CL].

[12] *Papers with code - imagenet benchmark (image classification)*. URL: https://paperswithcode.com/sota/image-classification-on-imagenet.

[13] Sachin Kumar, Prayag Tiwari, and Mikhail Zymbler. “Internet of Things is a revolutionary approach for future technology enhancement: a review”. In: *Journal of Big data* 6.1 (2019), pp. 1–21.

[14] Jiahui Yu et al. “CoCa: Contrastive Captioners are Image-Text Foundation Models”. In: (2022). DOI: 10.48550/ARXIV.2205.01917. URL: https://arxiv.org/abs/2205.01917.

[15] Christian Szegedy et al. *Rethinking the Inception Architecture for Computer Vision*. 2015. arXiv: 1512.00567 [cs.CV].

[16] Kaiming He et al. “Deep Residual Learning for Image Recognition”. In: *CoRR* abs/1512.03385 (2015). arXiv: 1512.03385. URL: http://arxiv.org/abs/1512.03385.

[17] Karen Simonyan and Andrew Zisserman. *Very Deep Convolutional Networks for Large-Scale Image Recognition*. 2015. arXiv: 1409.1556 [cs.CV].

[18] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. “ImageNet Classification with Deep Convolutional Neural Networks”. In: *Advances in Neural Information Processing Systems*. Ed. by F. Pereira et al. Vol. 25. Curran Associates, Inc., 2012. URL: https://proceedings.neurips.cc/paper/2012/file/c399862d3b9d6b76c8436e924a68c45b-Paper.pdf.

[19] Itay Hubara et al. “Quantized neural networks: Training neural networks with low precision weights and activations”. In: *The Journal of Machine Learning Research* 18.1 (2017), pp. 6869–6898.

[20] Mohammad Rastegari et al. *XNOR-Net: ImageNet Classification Using Binary Convolutional Neural Networks*. 2016. DOI: 10.48550/ARXIV.1603.05279. URL: https://arxiv.org/abs/1603.05279.

[21] Neil Houlsby et al. *Parameter-Efficient Transfer Learning for NLP*. 2019. DOI: 10.48550/ARXIV.1902.00751. URL: https://arxiv.org/abs/1902.00751.

[22] Jonas Pfeiffer et al. *AdapterHub: A Framework for Adapting Transformers*. 2020. DOI: 10.48550/ARXIV.2007.07779. URL: https://arxiv.org/abs/2007.07779.

[23] Alex Krizhevsky and Geoffrey Hinton. “Learning multiple layers of features from tiny images”. In: 0 (2009).

[24] Maria-Elena Nilsback and Andrew Zisserman. “Automated Flower Classification over a Large Number of Classes”. In: *Indian Conference on Computer Vision, Graphics and Image Processing*. Dec. 2008.

[25] Omkar M. Parkhi et al. “Cats and Dogs”. In: *IEEE Conference on Computer Vision and Pattern Recognition*. 2012.

[26] Adam Paszke et al. “PyTorch: An Imperative Style, High-Performance Deep Learning Library”. In: *Advances in Neural Information Processing Systems 32*. Curran Associates, Inc., 2019, pp. 8024–8035. URL: http://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf.

[27] Lukas Biewald. *Experiment Tracking with Weights and Biases*. Software available from wandb.com. 2020. URL: https://www.wandb.com/.

[28] Olga Russakovsky et al. “ImageNet Large Scale Visual Recognition Challenge”. In: *International Journal of Computer Vision (IJCV)* 115.3 (2015), pp. 211–252. DOI: 10.1007/s11263-015-0816-y.

[29] Allan Pinkus. “Approximation theory of the MLP model in neural networks”. In: *Acta Numerica* 8 (1999), pp. 143–195. DOI: 10.1017/S0962492900002919.

[30] Michael A. Nielsen. *Neural Networks and Deep Learning*. misc. 2018. URL: http://neuralnetworksanddeeplearning.com/.

[31] Leslie N. Smith. *A disciplined approach to neural network hyper-parameters: Part 1 – learning rate, batch size, momentum, and weight decay*. 2018. arXiv: 1803.09820 [cs.LG].

[32] Frank Rosenblatt. “The perceptron: a probabilistic model for information storage and organization in the brain.” In: *Psychological review* 65.6 (1958), p. 386.

[33] Pieter-Tjerk De Boer et al. “A tutorial on the cross-entropy method”. In: *Annals of operations research* 134 (2005), pp. 19–67.

[34] David E Rumelhart, Geoffrey E Hinton, and Ronald J Williams. “Learning representations by back-propagating errors”. In: *nature* 323.6088 (1986), pp. 533–536.

[35] J. Kiefer and J. Wolfowitz. “Stochastic Estimation of the Maximum of a Regression Function”. In: *The Annals of Mathematical Statistics* 23.3 (1952), pp. 462–466. DOI: 10.1214/aoms/1177729392. URL: https://doi.org/10.1214/aoms/1177729392.

[36] Tong Yu and Hong Zhu. “Hyper-Parameter Optimization: A Review of Algorithms and Applications”. In: (2020). DOI: 10.48550/ARXIV.2003.05689. URL: https://arxiv.org/abs/2003.05689.

[37] Yann A. LeCun et al. “Efficient BackProp”. In: *Neural Networks: Tricks of the Trade: Second Edition*. Ed. by Grégoire Montavon, Geneviève B. Orr, and Klaus-Robert Müller. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 9–48. ISBN: 978-3-642-35289-8. DOI: 10.1007/978-3-642-35289-8_3. URL: https://doi.org/10.1007/978-3-642-35289-8_3.

[38] Abien Fred Agarap. “Deep Learning using Rectified Linear Units (ReLU)”. In: *CoRR* abs/1803.08375 (2018). arXiv: 1803.08375. URL: http://arxiv.org/abs/1803.08375.

[39] Sergey Ioffe and Christian Szegedy. “Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift”. In: *CoRR* abs/1502.03167 (2015). arXiv: 1502.03167. URL: http://arxiv.org/abs/1502.03167.

[40] Joseph Redmon et al. *You Only Look Once: Unified, Real-Time Object Detection*. 2016. arXiv: 1506.02640 [cs.CV].

[41] Yann LeCun and Yoshua Bengio. “Convolutional Networks for Images, Speech, and Time Series”. In: *The Handbook of Brain Theory and Neural Networks*. Cambridge, MA, USA: MIT Press, 1998, pp. 255–258. ISBN: 0262511029.

[42] Vincent Dumoulin and Francesco Visin. “A guide to convolution arithmetic for deep learning”. In: *ArXiv e-prints* (Mar. 2016). eprint: 1603.07285.

[43] Matthew D Zeiler and Rob Fergus. “Visualizing and Understanding Convolutional Networks”. In: (2013). DOI: 10.48550/ARXIV.1311.2901. URL: https://arxiv.org/abs/1311.2901.

[44] Dhiraj Kalamkar et al. *A Study of BFLOAT16 for Deep Learning Training*. 2019. DOI: 10.48550/ARXIV.1905.12322. URL: https://arxiv.org/abs/1905.12322.

[45] Hao Wu et al. *Integer Quantization for Deep Learning Inference: Principles and Empirical Evaluation*. 2020. arXiv: 2004.09602 [cs.LG].

[46] Vincent Vanhoucke, Andrew Senior, and Mark Z Mao. “Improving the speed of neural networks on CPUs”. In: (2011).

[47] Xingang Wang et al. “Fast Object Detection Based on Binary Deep Convolution Neural Networks”. In: *CAAI Transactions on Intelligence Technology* 3 (Oct. 2018). DOI: 10.1049/trit.2018.1026.

[48] Jonathan Frankle and Michael Carbin. *The Lottery Ticket Hypothesis: Finding Sparse, Trainable Neural Networks*. 2019. arXiv: 1803.03635 [cs.LG].

[49] Chuanqi Tan et al. *A Survey on Deep Transfer Learning*. 2018. DOI: 10.48550/ARXIV.1808.01974. URL: https://arxiv.org/abs/1808.01974.

[50] Hoo-Chang Shin et al. “Deep convolutional neural networks for computer-aided detection: CNN architectures, dataset characteristics and transfer learning”. In: *IEEE transactions on medical imaging* 35.5 (2016), pp. 1285–1298.

[51] Chuanqi Tan et al. “A survey on deep transfer learning”. In: *Artificial Neural Networks and Machine Learning–ICANN 2018: 27th International Conference on Artificial Neural Networks, Rhodes, Greece, October 4-7, 2018, Proceedings, Part III 27*. Springer. 2018, pp. 270–279.

[52] Zechun Liu et al. *Bi-Real Net: Enhancing the Performance of 1-bit CNNs With Improved Representational Capability and Advanced Training Algorithm*. 2018. arXiv: 1808.00278 [cs.CV].

[53] Sylvestre-Alvise Rebuffi, Hakan Bilen, and Andrea Vedaldi. “Efficient parametrization of multi-domain deep neural networks”. In: *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*. 2018, pp. 8119–8127.

[54] Rodrigo Berriel et al. “Budget-Aware Adapters for Multi-Domain Learning”. In: *2019 IEEE/CVF International Conference on Computer Vision (ICCV)*. IEEE, Oct. 2019. DOI: 10.1109/iccv.2019.00047. URL: https://doi.org/10.1109%2Ficcv.2019.00047.

[55] Andrew G. Howard et al. *MobileNets: Efficient Convolutional Neural Networks for Mobile Vision Applications*. 2017. DOI: 10.48550/ARXIV.1704.04861. URL: https://arxiv.org/abs/1704.04861.

[56] Jia Deng et al. “Imagenet: A large-scale hierarchical image database”. In: *2009 IEEE conference on computer vision and pattern recognition*. Ieee. 2009, pp. 248–255.

[57] Diederik P. Kingma and Jimmy Ba. *Adam: A Method for Stochastic Optimization*. 2017. arXiv: 1412.6980 [cs.LG].

[58] Edward J. Hu et al. *LoRA: Low-Rank Adaptation of Large Language Models*. 2021. arXiv: 2106.09685 [cs.CL].

[59] Sean Fox et al. “Training Deep Neural Networks in Low-Precision with High Accuracy Using FPGAs”. In: *2019 International Conference on Field-Programmable Technology (ICFPT)*. 2019, pp. 1–9. DOI: 10.1109/ICFPT47387.2019.00009.

Chapter 9

Appendix

9.1 Base Networks

9.1.1 STE BNN

The following STE BNN is from Courbariaux et al. and was retrieved from `https://github.com/itayhubara/BinaryNet.pytorch.git`

Table 9.1: Summary Of Network Structure for STE BNN.

Layer (type)	Input Shape	Output Shape	Param #	Kernel Shape
ResNet_cifar10	[1, 3, 32, 32]	[1, 10]	–	–
+ BinarizeConv2d	[1, 3, 32, 32]	[1, 80, 32, 32]	2,160	[3, 3]
+ BatchNorm2d	[1, 80, 32, 32]	[1, 80, 32, 32]	160	–
+ Hardtanh	[1, 80, 32, 32]	[1, 80, 32, 32]	–	–
+ Sequential	[1, 80, 32, 32]	[1, 80, 32, 32]	–	–
| + BasicBlock	[1, 80, 32, 32]	[1, 80, 32, 32]	–	–
| | + BinarizeConv2d	[1, 80, 32, 32]	[1, 80, 32, 32]	57,600	[3, 3]
| | + BatchNorm2d	[1, 80, 32, 32]	[1, 80, 32, 32]	160	–
| | + Hardtanh	[1, 80, 32, 32]	[1, 80, 32, 32]	–	–
| | + BinarizeConv2d	[1, 80, 32, 32]	[1, 80, 32, 32]	57,600	[3, 3]
| | + BatchNorm2d	[1, 80, 32, 32]	[1, 80, 32, 32]	160	–
| | + Hardtanh	[1, 80, 32, 32]	[1, 80, 32, 32]	–	–
| + BasicBlock	[1, 80, 32, 32]	[1, 80, 32, 32]	–	–
| | + BinarizeConv2d	[1, 80, 32, 32]	[1, 80, 32, 32]	57,600	[3, 3]
| | + BatchNorm2d	[1, 80, 32, 32]	[1, 80, 32, 32]	160	–
| | + Hardtanh	[1, 80, 32, 32]	[1, 80, 32, 32]	–	–
| | + BinarizeConv2d	[1, 80, 32, 32]	[1, 80, 32, 32]	57,600	[3, 3]
| | + BatchNorm2d	[1, 80, 32, 32]	[1, 80, 32, 32]	160	–
| | + Hardtanh	[1, 80, 32, 32]	[1, 80, 32, 32]	–	–

Table 9.1: Summary Of Network Structure for STE BNN.

Layer (type)	Input Shape	Output Shape	Param #	Kernel Shape
+ Sequential	[1, 80, 32, 32]	[1, 160, 16, 16]	–	–
| + BasicBlock	[1, 80, 32, 32]	[1, 160, 16, 16]	–	–
| | + BinarizeConv2d	[1, 80, 32, 32]	[1, 160, 16, 16]	115,200	[3, 3]
| | + BatchNorm2d	[1, 160, 16, 16]	[1, 160, 16, 16]	320	–
| | + Hardtanh	[1, 160, 16, 16]	[1, 160, 16, 16]	–	–
| | + BinarizeConv2d	[1, 160, 16, 16]	[1, 160, 16, 16]	230,400	[3, 3]
| | + BatchNorm2d	[1, 160, 16, 16]	[1, 160, 16, 16]	320	–
| | + Hardtanh	[1, 160, 16, 16]	[1, 160, 16, 16]	–	–
| | + Sequential	[1, 80, 32, 32]	[1, 160, 16, 16]	13,120	–
| + BasicBlock	[1, 160, 16, 16]	[1, 160, 16, 16]	–	–
| | + BinarizeConv2d	[1, 160, 16, 16]	[1, 160, 16, 16]	230,400	[3, 3]
| | + BatchNorm2d	[1, 160, 16, 16]	[1, 160, 16, 16]	320	–
| | + Hardtanh	[1, 160, 16, 16]	[1, 160, 16, 16]	–	–
| | + BinarizeConv2d	[1, 160, 16, 16]	[1, 160, 16, 16]	230,400	[3, 3]
| | + BatchNorm2d	[1, 160, 16, 16]	[1, 160, 16, 16]	320	–
| | + Hardtanh	[1, 160, 16, 16]	[1, 160, 16, 16]	–	–
+ Sequential	[1, 160, 16, 16]	[1, 320, 8, 8]	–	–
| + BasicBlock	[1, 160, 16, 16]	[1, 320, 8, 8]	–	–
| | + BinarizeConv2d	[1, 160, 16, 16]	[1, 320, 8, 8]	460,800	[3, 3]
| | + BatchNorm2d	[1, 320, 8, 8]	[1, 320, 8, 8]	640	–
| | + Hardtanh	[1, 320, 8, 8]	[1, 320, 8, 8]	–	–
| | + BinarizeConv2d	[1, 320, 8, 8]	[1, 320, 8, 8]	921,600	[3, 3]
| | + BatchNorm2d	[1, 320, 8, 8]	[1, 320, 8, 8]	640	–
| | + Hardtanh	[1, 320, 8, 8]	[1, 320, 8, 8]	–	–
| | + Sequential	[1, 160, 16, 16]	[1, 320, 8, 8]	51,840	–
| + BasicBlock	[1, 320, 8, 8]	[1, 320, 8, 8]	–	–
| | + BinarizeConv2d	[1, 320, 8, 8]	[1, 320, 8, 8]	921,600	[3, 3]
| | + BatchNorm2d	[1, 320, 8, 8]	[1, 320, 8, 8]	640	–
| | + Hardtanh	[1, 320, 8, 8]	[1, 320, 8, 8]	–	–
| | + BinarizeConv2d	[1, 320, 8, 8]	[1, 320, 8, 8]	921,600	[3, 3]
| | + BatchNorm2d	[1, 320, 8, 8]	[1, 320, 8, 8]	640	–
| | + Hardtanh	[1, 320, 8, 8]	[1, 320, 8, 8]	–	–
+ AvgPool2d	[1, 320, 8, 8]	[1, 320, 1, 1]	–	8
+ BatchNorm1d	[1, 320]	[1, 320]	640	–
+ Hardtanh	[1, 320]	[1, 320]	–	–
+ BinarizeLinear	[1, 320]	[1, 10]	3,210	–

Table 9.1: Summary Of Network Structure for STE BNN.

Layer (type)	Input Shape	Output Shape	Param #	Kernel Shape
+ BatchNorm1d	[1, 10]	[1, 10]	20	–
+ LogSoftmax	[1, 10]	[1, 10]	–	–

9.1.2 ReAct Resnet18

The following Network is from Liu et al. [4] and can be found at`https://github.com/liuzechun/ReActNet.git`

Table 9.2: Weights and Summary of ReactNet used. The layers are ordered from input - output layer order

Layer (Key :depth-idx)	Input Shape	Output Shape	Param #	Kernel Shape
BiRealNet	[1, 3, 224, 224]	[1, 1000]	–	–
l-Conv2d: 1-1	[1, 3, 224, 224]	[1, 64, 112, 112]	9,408	[7, 7]
l-BatchNorm2d: 1-2	[1, 64, 112, 112]	[1, 64, 112, 112]	128	–
l-MaxPool2d: 1-3	[1, 64, 112, 112]	[1, 64, 56, 56]	–	3
l-Sequential: 1-4	[1, 64, 56, 56]	[1, 64, 56, 56]	–	–
l –BasicBlock: 2-1	[1, 64, 56, 56]	[1, 64, 56, 56]	–	–
l l –LearnableBias: 3-1	[1, 64, 56, 56]	[1, 64, 56, 56]	64	–
l l –BinaryActivation: 3-2	[1, 64, 56, 56]	[1, 64, 56, 56]	–	–
l l –HardBinaryConv: 3-3	[1, 64, 56, 56]	[1, 64, 56, 56]	36,928	[3, 3]
l l –BatchNorm2d: 3-4	[1, 64, 56, 56]	[1, 64, 56, 56]	128	–
l l –LearnableBias: 3-5	[1, 64, 56, 56]	[1, 64, 56, 56]	64	–
l l –PReLU: 3-6	[1, 64, 56, 56]	[1, 64, 56, 56]	64	–
l l –LearnableBias: 3-7	[1, 64, 56, 56]	[1, 64, 56, 56]	64	–
l –BasicBlock: 2-2	[1, 64, 56, 56]	[1, 64, 56, 56]	–	–
l l –LearnableBias: 3-8	[1, 64, 56, 56]	[1, 64, 56, 56]	64	–
l l –BinaryActivation: 3-9	[1, 64, 56, 56]	[1, 64, 56, 56]	–	–
l l –HardBinaryConv: 3-10	[1, 64, 56, 56]	[1, 64, 56, 56]	36,928	[3, 3]
l l –BatchNorm2d: 3-11	[1, 64, 56, 56]	[1, 64, 56, 56]	128	–
l l –LearnableBias: 3-12	[1, 64, 56, 56]	[1, 64, 56, 56]	64	–
l l –PReLU: 3-13	[1, 64, 56, 56]	[1, 64, 56, 56]	64	–
l l –LearnableBias: 3-14	[1, 64, 56, 56]	[1, 64, 56, 56]	64	–
l –BasicBlock: 2-3	[1, 64, 56, 56]	[1, 64, 56, 56]	–	–
l l –LearnableBias: 3-15	[1, 64, 56, 56]	[1, 64, 56, 56]	64	–
l l –BinaryActivation: 3-16	[1, 64, 56, 56]	[1, 64, 56, 56]	–	–
l l –HardBinaryConv: 3-17	[1, 64, 56, 56]	[1, 64, 56, 56]	36,928	[3, 3]

Table 9.2: Weights and Summary of ReactNet used. The layers are ordered from input - output layer order

Layer (Key :depth-idx)	Input Shape	Output Shape	Param #	Kernel Shape
| | –BatchNorm2d: 3-18	[1, 64, 56, 56]	[1, 64, 56, 56]	128	–
| | –LearnableBias: 3-19	[1, 64, 56, 56]	[1, 64, 56, 56]	64	–
| | –PReLU: 3-20	[1, 64, 56, 56]	[1, 64, 56, 56]	64	–
| | –LearnableBias: 3-21	[1, 64, 56, 56]	[1, 64, 56, 56]	64	–
| –BasicBlock: 2-4	[1, 64, 56, 56]	[1, 64, 56, 56]	–	–
| | –LearnableBias: 3-22	[1, 64, 56, 56]	[1, 64, 56, 56]	64	–
| | –BinaryActivation: 3-23	[1, 64, 56, 56]	[1, 64, 56, 56]	–	–
| | –HardBinaryConv: 3-24	[1, 64, 56, 56]	[1, 64, 56, 56]	36,928	[3, 3]
| | –BatchNorm2d: 3-25	[1, 64, 56, 56]	[1, 64, 56, 56]	128	–
| | –LearnableBias: 3-26	[1, 64, 56, 56]	[1, 64, 56, 56]	64	–
| | –PReLU: 3-27	[1, 64, 56, 56]	[1, 64, 56, 56]	64	–
| | –LearnableBias: 3-28	[1, 64, 56, 56]	[1, 64, 56, 56]	64	–
|-Sequential: 1-5	[1, 64, 56, 56]	[1, 128, 28, 28]	–	–
| –BasicBlock: 2-5	[1, 64, 56, 56]	[1, 128, 28, 28]	–	–
| | –LearnableBias: 3-29	[1, 64, 56, 56]	[1, 64, 56, 56]	64	–
| | –BinaryActivation: 3-30	[1, 64, 56, 56]	[1, 64, 56, 56]	–	–
| | –HardBinaryConv: 3-31	[1, 64, 56, 56]	[1, 128, 28, 28]	73,856	[3, 3]
| | –BatchNorm2d: 3-32	[1, 128, 28, 28]	[1, 128, 28, 28]	256	–
| | –Sequential: 3-33	[1, 64, 56, 56]	[1, 128, 28, 28]	8,448	–
| | –LearnableBias: 3-34	[1, 128, 28, 28]	[1, 128, 28, 28]	128	–
| | –PReLU: 3-35	[1, 128, 28, 28]	[1, 128, 28, 28]	128	–
| | –LearnableBias: 3-36	[1, 128, 28, 28]	[1, 128, 28, 28]	128	–
| –BasicBlock: 2-6	[1, 128, 28, 28]	[1, 128, 28, 28]	–	–
| | –LearnableBias: 3-37	[1, 128, 28, 28]	[1, 128, 28, 28]	128	–
| | –BinaryActivation: 3-38	[1, 128, 28, 28]	[1, 128, 28, 28]	–	–
| | –HardBinaryConv: 3-39	[1, 128, 28, 28]	[1, 128, 28, 28]	147,584	[3, 3]
| | –BatchNorm2d: 3-40	[1, 128, 28, 28]	[1, 128, 28, 28]	256	–
| | –LearnableBias: 3-41	[1, 128, 28, 28]	[1, 128, 28, 28]	128	–
| | –PReLU: 3-42	[1, 128, 28, 28]	[1, 128, 28, 28]	128	–
| | –LearnableBias: 3-43	[1, 128, 28, 28]	[1, 128, 28, 28]	128	–
| –BasicBlock: 2-7	[1, 128, 28, 28]	[1, 128, 28, 28]	–	–
| | –LearnableBias: 3-44	[1, 128, 28, 28]	[1, 128, 28, 28]	128	–
| | –BinaryActivation: 3-45	[1, 128, 28, 28]	[1, 128, 28, 28]	–	–
| | –HardBinaryConv: 3-46	[1, 128, 28, 28]	[1, 128, 28, 28]	147,584	[3, 3]
| | –BatchNorm2d: 3-47	[1, 128, 28, 28]	[1, 128, 28, 28]	256	–

Table 9.2: Weights and Summary of ReactNet used. The layers are ordered from input - output layer order

Layer (Key :depth-idx)	Input Shape	Output Shape	Param #	Kernel Shape
| | –LearnableBias: 3-48	[1, 128, 28, 28]	[1, 128, 28, 28]	128	–
| | –PReLU: 3-49	[1, 128, 28, 28]	[1, 128, 28, 28]	128	–
| | –LearnableBias: 3-50	[1, 128, 28, 28]	[1, 128, 28, 28]	128	–
| –BasicBlock: 2-8	[1, 128, 28, 28]	[1, 128, 28, 28]	–	–
| | –LearnableBias: 3-51	[1, 128, 28, 28]	[1, 128, 28, 28]	128	–
| | –BinaryActivation: 3-52	[1, 128, 28, 28]	[1, 128, 28, 28]	–	–
| | –HardBinaryConv: 3-53	[1, 128, 28, 28]	[1, 128, 28, 28]	147,584	[3, 3]
| | –BatchNorm2d: 3-54	[1, 128, 28, 28]	[1, 128, 28, 28]	256	–
| | –LearnableBias: 3-55	[1, 128, 28, 28]	[1, 128, 28, 28]	128	–
| | –PReLU: 3-56	[1, 128, 28, 28]	[1, 128, 28, 28]	128	–
| | –LearnableBias: 3-57	[1, 128, 28, 28]	[1, 128, 28, 28]	128	–
|-Sequential: 1-6	[1, 128, 28, 28]	[1, 256, 14, 14]	–	–
| –BasicBlock: 2-9	[1, 128, 28, 28]	[1, 256, 14, 14]	–	–
| | –LearnableBias: 3-58	[1, 128, 28, 28]	[1, 128, 28, 28]	128	–
| | –BinaryActivation: 3-59	[1, 128, 28, 28]	[1, 128, 28, 28]	–	–
| | –HardBinaryConv: 3-60	[1, 128, 28, 28]	[1, 256, 14, 14]	295,168	[3, 3]
| | –BatchNorm2d: 3-61	[1, 256, 14, 14]	[1, 256, 14, 14]	512	–
| | –Sequential: 3-62	[1, 128, 28, 28]	[1, 256, 14, 14]	33,280	–
| | –LearnableBias: 3-63	[1, 256, 14, 14]	[1, 256, 14, 14]	256	–
| | –PReLU: 3-64	[1, 256, 14, 14]	[1, 256, 14, 14]	256	–
| | –LearnableBias: 3-65	[1, 256, 14, 14]	[1, 256, 14, 14]	256	–
| –BasicBlock: 2-10	[1, 256, 14, 14]	[1, 256, 14, 14]	–	–
| | –LearnableBias: 3-66	[1, 256, 14, 14]	[1, 256, 14, 14]	256	–
| | –BinaryActivation: 3-67	[1, 256, 14, 14]	[1, 256, 14, 14]	–	–
| | –HardBinaryConv: 3-68	[1, 256, 14, 14]	[1, 256, 14, 14]	590,080	[3, 3]
| | –BatchNorm2d: 3-69	[1, 256, 14, 14]	[1, 256, 14, 14]	512	–
| | –LearnableBias: 3-70	[1, 256, 14, 14]	[1, 256, 14, 14]	256	–
| | –PReLU: 3-71	[1, 256, 14, 14]	[1, 256, 14, 14]	256	–
| | –LearnableBias: 3-72	[1, 256, 14, 14]	[1, 256, 14, 14]	256	–
| –BasicBlock: 2-11	[1, 256, 14, 14]	[1, 256, 14, 14]	–	–
| | –LearnableBias: 3-73	[1, 256, 14, 14]	[1, 256, 14, 14]	256	–
| | –BinaryActivation: 3-74	[1, 256, 14, 14]	[1, 256, 14, 14]	–	–
| | –HardBinaryConv: 3-75	[1, 256, 14, 14]	[1, 256, 14, 14]	590,080	[3, 3]
| | –BatchNorm2d: 3-76	[1, 256, 14, 14]	[1, 256, 14, 14]	512	–
| | –LearnableBias: 3-77	[1, 256, 14, 14]	[1, 256, 14, 14]	256	–

Table 9.2: Weights and Summary of ReactNet used. The layers are ordered from input - output layer order

Layer (Key :depth-idx)	Input Shape	Output Shape	Param #	Kernel Shape
| | –PReLU: 3-78	[1, 256, 14, 14]	[1, 256, 14, 14]	256	–
| | –LearnableBias: 3-79	[1, 256, 14, 14]	[1, 256, 14, 14]	256	–
| –BasicBlock: 2-12	[1, 256, 14, 14]	[1, 256, 14, 14]	–	–
| | –LearnableBias: 3-80	[1, 256, 14, 14]	[1, 256, 14, 14]	256	–
| | –BinaryActivation: 3-81	[1, 256, 14, 14]	[1, 256, 14, 14]	–	–
| | –HardBinaryConv: 3-82	[1, 256, 14, 14]	[1, 256, 14, 14]	590,080	[3, 3]
| | –BatchNorm2d: 3-83	[1, 256, 14, 14]	[1, 256, 14, 14]	512	–
| | –LearnableBias: 3-84	[1, 256, 14, 14]	[1, 256, 14, 14]	256	–
| | –PReLU: 3-85	[1, 256, 14, 14]	[1, 256, 14, 14]	256	–
| | –LearnableBias: 3-86	[1, 256, 14, 14]	[1, 256, 14, 14]	256	–
|-Sequential: 1-7	[1, 256, 14, 14]	[1, 512, 7, 7]	–	–
| –BasicBlock: 2-13	[1, 256, 14, 14]	[1, 512, 7, 7]	–	–
| | –LearnableBias: 3-87	[1, 256, 14, 14]	[1, 256, 14, 14]	256	–
| | –BinaryActivation: 3-88	[1, 256, 14, 14]	[1, 256, 14, 14]	–	–
| | –HardBinaryConv: 3-89	[1, 256, 14, 14]	[1, 512, 7, 7]	1,180,160	[3, 3]
| | –BatchNorm2d: 3-90	[1, 512, 7, 7]	[1, 512, 7, 7]	1,024	–
| | –Sequential: 3-91	[1, 256, 14, 14]	[1, 512, 7, 7]	132,096	–
| | –LearnableBias: 3-92	[1, 512, 7, 7]	[1, 512, 7, 7]	512	–
| | –PReLU: 3-93	[1, 512, 7, 7]	[1, 512, 7, 7]	512	–
| | –LearnableBias: 3-94	[1, 512, 7, 7]	[1, 512, 7, 7]	512	–
| –BasicBlock: 2-14	[1, 512, 7, 7]	[1, 512, 7, 7]	–	–
| | –LearnableBias: 3-95	[1, 512, 7, 7]	[1, 512, 7, 7]	512	–
| | –BinaryActivation: 3-96	[1, 512, 7, 7]	[1, 512, 7, 7]	–	–
| | –HardBinaryConv: 3-97	[1, 512, 7, 7]	[1, 512, 7, 7]	2,359,808	[3, 3]
| | –BatchNorm2d: 3-98	[1, 512, 7, 7]	[1, 512, 7, 7]	1,024	–
| | –LearnableBias: 3-99	[1, 512, 7, 7]	[1, 512, 7, 7]	512	–
| | –PReLU: 3-100	[1, 512, 7, 7]	[1, 512, 7, 7]	512	–
| | –LearnableBias: 3-101	[1, 512, 7, 7]	[1, 512, 7, 7]	512	–
| –BasicBlock: 2-15	[1, 512, 7, 7]	[1, 512, 7, 7]	–	–
| | –LearnableBias: 3-102	[1, 512, 7, 7]	[1, 512, 7, 7]	512	–
| | –BinaryActivation: 3-103	[1, 512, 7, 7]	[1, 512, 7, 7]	–	–
| | –HardBinaryConv: 3-104	[1, 512, 7, 7]	[1, 512, 7, 7]	2,359,808	[3, 3]
| | –BatchNorm2d: 3-105	[1, 512, 7, 7]	[1, 512, 7, 7]	1,024	–
| | –LearnableBias: 3-106	[1, 512, 7, 7]	[1, 512, 7, 7]	512	–
| | –PReLU: 3-107	[1, 512, 7, 7]	[1, 512, 7, 7]	512	–

Table 9.2: Weights and Summary of ReactNet used. The layers are ordered from input - output layer order

Layer (Key :depth-idx)	Input Shape	Output Shape	Param #	Kernel Shape
‖ –LearnableBias: 3-108	[1, 512, 7, 7]	[1, 512, 7, 7]	512	–
‖ –BasicBlock: 2-16	[1, 512, 7, 7]	[1, 512, 7, 7]	–	–
‖ ‖ –LearnableBias: 3-109	[1, 512, 7, 7]	[1, 512, 7, 7]	512	–
‖ ‖ –BinaryActivation: 3-110	[1, 512, 7, 7]	[1, 512, 7, 7]	–	–
‖ ‖ –HardBinaryConv: 3-111	[1, 512, 7, 7]	[1, 512, 7, 7]	2,359,808	[3, 3]
‖ ‖ –BatchNorm2d: 3-112	[1, 512, 7, 7]	[1, 512, 7, 7]	1,024	–
‖ ‖ –LearnableBias: 3-113	[1, 512, 7, 7]	[1, 512, 7, 7]	512	–
‖ ‖ –PReLU: 3-114	[1, 512, 7, 7]	[1, 512, 7, 7]	512	–
‖ ‖ –LearnableBias: 3-115	[1, 512, 7, 7]	[1, 512, 7, 7]	512	–
‖-AdaptiveAvgPool2d: 1-8	[1, 512, 7, 7]	[1, 512, 1, 1]	–	–
‖-Linear: 1-9	[1, 512]	[1, 1000]	513,000	–

9.2 Adapters

9.2.1 Serial Adapters

Table 9.3: Summary of Adapter Architect for serial adapters

Location 1	Input Shape	Output Shape	Param #	Kernel Shape
conv_channel_adapter3	[1, 80, 32, 32]	[1, 80, 32, 32]	–	–
+ BatchNorm2d	[1, 80, 32, 32]	[1, 80, 32, 32]	160	–
+ Conv2d	[1, 80, 32, 32]	[1, 80, 32, 32]	6,400	[1, 1]
+ BatchNorm2d	[1, 80, 32, 32]	[1, 80, 32, 32]	160	–

Location 2	Input Shape	Output Shape	Param #	Kernel Shape
conv_channel_adapter3	[1, 160, 16, 16]	[1, 160, 16, 16]	–	–
+ BatchNorm2d	[1, 160, 16, 16]	[1, 160, 16, 16]	320	–
+ Conv2d	[1, 160, 16, 16]	[1, 160, 16, 16]	25,600	[1, 1]
+ BatchNorm2d	[1, 160, 16, 16]	[1, 160, 16, 16]	320	–

Location 3	Input Shape	Output Shape	Param #	Kernel Shape
conv_channel_adapter3	[1, 320, 8, 8]	[1, 320, 8, 8]	–	–

Table 9.3: Summary of Adapter Architect for serial adapters

Location 1	Input Shape	Output Shape	Param #	Kernel Shape
+ BatchNorm2d	[1, 320, 8, 8]	[1, 320, 8, 8]	640	–
+ Conv2d	[1, 320, 8, 8]	[1, 320, 8, 8]	102,400	[1, 1]
+ BatchNorm2d	[1, 320, 8, 8]	[1, 320, 8, 8]	640	–

9.2.2 Parallel Thinblock Adapters

9.2.2.1 STE BNN UDTA Adapter

Table 9.4: Network Summary of Custom thinblock adapter before Pruning or grouped convolutions For STE BNN

Layer (type)	Input Shape	Output Shape	Param #	Kernel Shape
uniAdapt_Net	[1, 80, 32, 32]	[1, 320, 1, 1]	–	–
+ thinBlock3	[1, 80, 32, 32]	[1, 160, 32, 32]	–	–
| + Conv2d	[1, 80, 32, 32]	[1, 80, 32, 32]	720	[3, 3]
| + BatchNorm2d	[1, 80, 32, 32]	[1, 80, 32, 32]	160	–
| + Conv2d	[1, 80, 32, 32]	[1, 160, 32, 32]	12,800	[1, 1]
| + BatchNorm2d	[1, 160, 32, 32]	[1, 160, 32, 32]	320	–
+ AdaptiveAvgPool2d	[1, 160, 32, 32]	[1, 160, 16, 16]	–	–
+ thinBlock3	[1, 160, 16, 16]	[1, 320, 16, 16]	–	–
| + Conv2d	[1, 160, 16, 16]	[1, 160, 16, 16]	1,440	[3, 3]
| + BatchNorm2d	[1, 160, 16, 16]	[1, 160, 16, 16]	320	–
| + Conv2d	[1, 160, 16, 16]	[1, 320, 16, 16]	51,200	[1, 1]
| + BatchNorm2d	[1, 320, 16, 16]	[1, 320, 16, 16]	640	–
+ AdaptiveAvgPool2d	[1, 320, 16, 16]	[1, 320, 8, 8]	–	–
+ thinBlock3	[1, 320, 8, 8]	[1, 320, 8, 8]	–	–
| + Conv2d	[1, 320, 8, 8]	[1, 320, 8, 8]	2,880	[3, 3]
| + BatchNorm2d	[1, 320, 8, 8]	[1, 320, 8, 8]	640	–
| + Conv2d	[1, 320, 8, 8]	[1, 320, 8, 8]	102,400	[1, 1]
| + BatchNorm2d	[1, 320, 8, 8]	[1, 320, 8, 8]	640	–
+ AdaptiveAvgPool2d	[1, 320, 8, 8]	[1, 320, 1, 1]	–	–

9.2.2.2 ReActUDTA Adapter

Table 9.5: Network Summary of Custom thinblock adapter before Pruning or grouped convolutions

Layer (type)	Input Shape	Output Shape	Param #	Kernel Shape
uniAdapt_Net_React	[1, 64, 56, 56]	[1, 512]	–	–
+ thinBlock3	[1, 64, 56, 56]	[1, 128, 56, 56]	–	–
| + Conv2d	[1, 64, 56, 56]	[1, 64, 56, 56]	576	[3, 3]
| + BatchNorm2d	[1, 64, 56, 56]	[1, 64, 56, 56]	128	–
| + Conv2d	[1, 64, 56, 56]	[1, 128, 56, 56]	8,192	[1, 1]
| + BatchNorm2d	[1, 128, 56, 56]	[1, 128, 56, 56]	256	–
+ AdaptiveAvgPool2d	[1, 128, 56, 56]	[1, 128, 28, 28]	–	–
+ thinBlock3	[1, 128, 28, 28]	[1, 256, 28, 28]	–	–
| + Conv2d	[1, 128, 28, 28]	[1, 128, 28, 28]	1,152	[3, 3]
| + BatchNorm2d	[1, 128, 28, 28]	[1, 128, 28, 28]	256	–
| + Conv2d	[1, 128, 28, 28]	[1, 256, 28, 28]	32,768	[1, 1]
| + BatchNorm2d	[1, 256, 28, 28]	[1, 256, 28, 28]	512	–
+ AdaptiveAvgPool2d	[1, 256, 28, 28]	[1, 256, 14, 14]	–	–
+ thinBlock3	[1, 256, 14, 14]	[1, 512, 14, 14]	–	–
| + Conv2d	[1, 256, 14, 14]	[1, 256, 14, 14]	2,304	[3, 3]
| + BatchNorm2d	[1, 256, 14, 14]	[1, 256, 14, 14]	512	–
| + Conv2d	[1, 256, 14, 14]	[1, 512, 14, 14]	131,072	[1, 1]
| + BatchNorm2d	[1, 512, 14, 14]	[1, 512, 14, 14]	1,024	–
+ AdaptiveAvgPool2d	[1, 512, 14, 14]	[1, 512, 7, 7]	–	–
+ thinBlock3	[1, 512, 7, 7]	[1, 512, 7, 7]	–	–
| + Conv2d	[1, 512, 7, 7]	[1, 512, 7, 7]	4,608	[3, 3]
| + BatchNorm2d	[1, 512, 7, 7]	[1, 512, 7, 7]	1,024	–
| + Conv2d	[1, 512, 7, 7]	[1, 512, 7, 7]	262,144	[1, 1]
| + BatchNorm2d	[1, 512, 7, 7]	[1, 512, 7, 7]	1,024	–
+ AdaptiveAvgPool2d	[1, 512, 7, 7]	[1, 512, 1, 1]	–	–

9.3 Custom Benchmarks

9.3.1 Cifar5-5

Cifar10 and Cifar100 can be obtained from `https://www.cs.toronto.edu/~kriz/cifar.html` [23]

9.3.1.1 Intial Training Dataset

Table 9.6: Cifar5-5 Initial Dataset STE BNN is trained on.

Class	Training Images Per Class	Test Images Per Class
airplane	5000	1000
automobile	5000	1000
bird	5000	1000
cat	5000	1000
deer	5000	1000
Total Images	25000	5000

9.3.1.2 Target Dataset

Table 9.7: Cifar5-5 Target Dataset STE BNN is Finetuned on.

Class	Training Images Per Class	Test Images Per Class
dog	5000	1000
frog	5000	1000
horse	5000	1000
ship	5000	1000
truck	5000	1000
Total images	25000	5000

9.3.2 Cifar80-20

9.3.2.1 Initial Training Dataset

[h]Table 9.8: Cifar80 Intitial Dataset STE BNN is trained on.

Class	Training Images Per Class	Test Images Per Class
chair	500	100
chimpanzee	500	100
clock	500	100
cloud	500	100
cockroach	500	100
couch	500	100
crab	500	100
crocodile	500	100
cup	500	100
dinosaur	500	100

dolphin	500	100
elephant	500	100
flatfish	500	100
forest	500	100
fox	500	100
girl	500	100
hamster	500	100
house	500	100
kangaroo	500	100
keyboard	500	100
lamp	500	100
lawn_mower	500	100
leopard	500	100
lion	500	100
lizard	500	100
lobster	500	100
man	500	100
maple_tree	500	100
motorcycle	500	100
mountain	500	100
mouse	500	100
mushroom	500	100
oak_tree	500	100
orange	500	100
orchid	500	100
otter	500	100
palm_tree	500	100
pear	500	100
pickup_truck	500	100
pine_tree	500	100
plain	500	100
plate	500	100
poppy	500	100
porcupine	500	100
possum	500	100
rabbit	500	100
raccoon	500	100
ray	500	100
road	500	100

rocket	500	100
rose	500	100
sea	500	100
seal	500	100
shark	500	100
shrew	500	100
skunk	500	100
skyscraper	500	100
snail	500	100
snake	500	100
spider	500	100
squirrel	500	100
streetcar	500	100
sunflower	500	100
sweet_pepper	500	100
table	500	100
tank	500	100
telephone	500	100
television	500	100
tiger	500	100
tractor	500	100
train	500	100
trout	500	100
tulip	500	100
turtle	500	100
wardrobe	500	100
whale	500	100
willow_tree	500	100
wolf	500	100
woman	500	100
worm	500	100
Total Images	40000	8000

9.3.2.2 Target Dataset

Table 9.9: Cifar20 Target Dataset STE BNN is finetuned on.

Class	**Training Images Per Class**	**Test Images Per Class**
apple	500	100

aquarium_fish	500	100
baby	500	100
bear	500	100
beaver	500	100
bed	500	100
bee	500	100
beetle	500	100
bicycle	500	100
bottle	500	100
bowl	500	100
boy	500	100
bridge	500	100
bus	500	100
butterfly	500	100
camel	500	100
can	500	100
castle	500	100
caterpillar	500	100
cattle	500	100
Total Images	10000	2000